BQCD Manual

T.R. Haar, Y. Nakamura and H. Stüben

June 2017

Taylor Ryan Haar
CSSM, Department of Physics
The University of Adelaide
Adelaide, SA
Australia 5005

<taylor.haar@adelaide.edu.au>

Yoshifumi Nakamura
RIKEN Advanced Institute for Computational Science
Kobe, Hyogo 650-0047
Japan

<nakamura@riken.jp>

Hinnerk Stüben
Universität Hamburg
Regionales Rechenzentrum
20146 Hamburg
Germany

<hinnerk.stueben@uni-hamburg.de>

$Id: svn_id.tex 989 2017-06-20 07:19:40Z rzxa001 $

Contents

Preface

BQCD is a Hybrid Monte-Carlo program for simulating lattice QCD with dynamical Wilson fermions. The development of BQCD was started in 1998 by H.S. for the two flavour case and the original Wilson action. It was written for a study of parallel tempering [3, 4]. At that time the whole parallelization framework was completed.

Two years later the program was extended in two directions. The first direction was the implementation of clover $O(a)$ improvement of the fermionic action. With the availability of clover improvement, BQCD became one of the main production codes of the QCDSF collaboration [5]. The second direction was the addition of an external field to the standard Wilson action in order to study the Aoki phase [18, 19]. The next milestone was the implementation of the Hasenbusch trick [6, 7].

Since 2006, the code was mainly developed by Y.N. He largely extended and improved the code to enable simulations including a third fermion flavour [8, 9, 10, 11], the hopping term with chemical potential [12], a CPT breaking term [13], measurement routines for rectangular plaquettes, Wilson flow, quark determinant, eigenvalues of the Dirac matrix as well as meson and baryon propagators, and state-of-the art solvers.

In 2013 QED was added by Y.N. and H.S. [14]. Portable SIMD vectorization was implemented by H.S. in 2015. In 2017, T.H. added polynomial filtering [15, 16] and improved the manual, while Y.N. added the axion [17].

The program has been used by several groups, e.g. the group of M. Mller-Preussker [18, 19], the DIK Collaboration [20, 21], QPACE [22], RQCD [23], Japanese finite density [12] and finite temperature [24] projects, as well as CSSM [25].

BQCD became free software under the *GNU General Public License* with its presentation at Lattice 2010 [1]. We hope that it will be useful for others and kindly ask users to cite our contribution to the proceedings of *Lattice 2017* [2] if the code is used to prepare a publication.

June 2017

Taylor Ryan Haar
Yoshifumi Nakamura
Hinnerk Stüben

1. Overview

BQCD is a Hybrid Monte Carlo program for generating configurations with a clover type action.

Implemented actions for dynamical simulations:

- Tree-level improved gauge actions
- Up to 6 distinct double-flavour pseudofermions and 6 single-flavour pseudofermions
- The clover action
- Stout link smearing
- Parity-flavour breaking
- Chemical potential
- CPT breaking
- QCD+QED
- The axion

Implemented algorithms for improving performance:

- RHMC algorithm
- Multi timescale integration, both nested and generalized
- Mass preconditioning
- Polynomial filtering
- Truncated RHMC
- Minimal norm integrators (Omelyan)
- Conjugate gradient with SIMD vectorization

After compiling the program (section 2), follow the quick start guide in section 3.1 to start using BQCD. The rest of section 3 explains how to use BQCD, which includes documentation on most of the input keywords (section 3.3). The following section 4 and section 5 explain the concepts. Then, section 6 explains how the code is implemented, and section 7 gives various ways to tune the program.

1.1. Summary of changes

- version : 5.1.0 (June 2017)
  - simulation of QCD+axion
  - simulation of QCD+QED
  - implementation of the conjugate gradient solver with SIMD intrinsics
  - polynomial filtering and truncated RHMC
  - the generalized integration scheme (see section 5.4)
- version : 4.1.0 (October 2011)
  - GCRO-DR solver
  - replay trick
  - Schrödinger functional method to determine c_{SW}
  - further RHMC tuning (see section 5.2)
  - SSE implementation of the hopping and clover matrix multiplications (set `libd = 103` in `Makefile.var`)
  - Check return value of functions for reading/writing ILDG format
  - Minor changes for printing and function interface
- version : 4.0.0 (June 2010)
  - first public version

2. Installation

2.1. Prerequisites

The default directory for finding prerequisite packages is

```
$HOME/opt/package
```

but this can be changed in the file `Makefile.in`. It is recommended to use the same compiler for building packages and BQCD.

2.1.1. lime

The *lime* library is needed for storing configurations in the ILDG format (International Lattice Data Grid, see: `http://plone.jldg.org/wiki`). It can be downloaded from:

```
http://usqcd.jlab.org/usqcd-software/c-lime/lime-1.3.2.tar.gz
```

Installation:

```
cd ~/opt
tar zxvf lime-1.3.2.tar.gz
cd lime-1.3.2
export CC=non-default-compiler            # optional
export CFLAGS=non-default-compiler-flags  # optional
./configure --prefix=$PWD
make
```

2.1.2. LAPACK and ScaLAPACK

LAPACK (Linear Algebra Package) and ScaLAPACK (Scalable LAPACK) are needed to get the full functionality. The code will compile without but will complain at runtime if they are needed but not provided. The original code can be downloaded from

```
http://www.netlib.org
```

but typically both libraries are installed on HPC systems. If an independent installation is used, `LAPACK` and `SCALAPACK` have to be defined in `Makefile.var`:

```
LAPACK = -L$(HOME)/opt/lapack -llapack -lrefblas -ltmglib
SCALAPACK = $(HOME)/opt/scalapack/libscalapack.a
```

Definitions for pre-installed libraries can be found in:

```
platform/Makefile-platform.var
```

2.2. Download

The source code of BQCD and this manual can be downloaded from:

```
https://www.rrz.uni-hamburg.de/bqcd
```

2.3. License

2.4. Configuration

2.4.1. Supported platforms

Platform dependent parts are kept in `Makefile.var` which is a symbolic link to a file in the `platform` directory

```
Makefile.var -> platform/Makefile-platform.var
```

for example:

```
Makefile.var -> platform/Makefile-gnu.var
```

One can prepare working on a particular platform by entering the command

```
make prep-platform
```

which creates the symbolic link, for example:

```
make prep-gnu
```

In the `platform` directory one can find files for machines that were used in the past but that are not necessarily up-to-date. Currently you can expect that compilation works in these cases:

`gnu`	GNU compiler, Open-MPI/MPICH
`hlrn3`	Cray XC40 (Intel compiler, Cray MPI)
`intel`	Intel compiler, Intel MPI
`juqueen`	IBM BlueGene/Q at JSC Jülich

2.4.2. Settings in `Makefile.var`

In `Makefile.var` one can make the following high level settings:

`timing`	= *empty* or `1`	switch on profiling
`mpi`	= *empty* or `1`	single processor program or MPI
`omp`	= *empty* or `1`	compile with OpenMP
`debug`	= *empty* or `1`	compile with debug flags
`libd`	= `100`	which hopping matrix multiplication
`random`	= `ranlux-3.2`	which random number generator

Based on these high level settings, several low level settings are made. This includes compiler flags, preprocessor flags and which libraries are used. For more information, see `platform/Makefile-EXPLAINED.var`.

The `make` procedure is not always straightforward. Instead of `make`,

```
make fast
```

just builds the binary `bqcd5`.

2.4.3. Configuring/porting SIMD

Configuring/porting SIMD is described in `cg/README`.

2.5. Testing

Reference test runs can be found in the `data/` directory. For each case there is an input and reference output file:

```
bqcd.NNN.input
bqcd.NNN.output
```

The test cases are described in comments:

```
grep comment *.input
```

A test run looks like this:

```
cd data
../bqcd5 bqcd.NNN.input bqcd.NNN.res
diff bqcd.NNN.output bqcd.NNN.res
```

Due to rounding differences the output will not be identical to the reference output but should be very close to the reference output.

In order to test a parallel run

- set the number of processes to be used for each lattice dimension in the input file, for example

  ```
  processes  1 1 2 4
  ```

  decomposes the z- into 2 domains and the t-direction into 4 domains,
- run BQCD on the appropriate number on processes (8 in the above example):

  ```
  mpirun -np 8 ../bqcd5 bqcd.NNN.output bqcd.NNN.res
  ```

  In principle the output (`bqcd.NNN.res`) is identical to the output from the sequential run (again up to rounding differences). This holds true for any decomposition.

3. Usage

3.1. Quickstart guide

To get started with BQCD, take a look the example input file `bqcd.300.input`, located in the `data/` directory. The input file is formatted as a series of 'keyword value' pairs, with the `#` character being used for comments.

3.1.1. Basics

Let's look at the first few keywords in this file:

```
comment                    "Test for Nf=2+1"
run                         300
lattice                       4    4    4    4
processes                     1    1    1    1
```

The `comment` is just that, and the `run` number is an identifier for this file's particular configuration.

The next few keywords describe how the lattice is set up. The keyword `lattice` sets up a $4^3 \times 4$ lattice, with the extents given the usual `xyzt` order. `processes` describes how the lattice is split between processes along each dimension, with the total number of processes equal to the product. In this case, there is just one process.

3.1.2. Gauge and fermion fields

Next are some keywords describing the fields on the lattice:

```
gauge_action               TREE
beta                       5.5
fermi_action               SLRC
csw                        2.65
n_stout                    1
alpha                      0.1
```

These give a tree-level improved gauge action with $\beta = 5.5$ and a Clover fermion action with $c_{SW} = 2.65$ and stout links on the Wilson part of the fermion action with 1 sweep and $\alpha = 0.1$. Refer to `gauge_action` and `fermi_action` for information on other choices of actions.

The keywords starting with `nf` determine which pseudofermions we have on this lattice:

```
nf2_kappa1                  0.121095
nf2_kappa1h1                0.1203
hmc_hkappa                  1

nf1_kappa1                  0.120512
nf1_kappa1npf               2
nf1_kappa1nth               4
```

This is a N_f=2+1 run with two kinds of pseudofermions:

- `nf2_kappa1` produces a double-flavour pseudofermion with hopping parameter $\kappa = 0.121095$, which has a Hasenbusch filter given by `nf2_kappa1h1` at $\kappa' = 0.1203$. `hmc_hkappa` toggles which form of Hasenbusch is used (κ' or $\kappa + \rho$).
- `nf1_kappa1` produces a single-flavour pseudofermion with $\kappa = 0.120512$ via RHMC, which is constructed via two pseudofermions (`nf1_kappa1npf`) with a rational approximation $R(M^\dagger M) \approx (M^\dagger M)^{-1/4}$ (`nf1_kappa1nth`).

3.1.3. Molecular dynamics

Now for some information about how the molecular dynamics part of HMC is integrated:

```
hmc_trajectory_length       1.0
hmc_integrator1             2MNSTS
hmc_integrator2             2MNSTS
hmc_integrator3             2MNSTS
hmc_integrator4             2MNSTS

hmc_steps                   5
hmc_m_scale                 2
hmc_m_scale2                2
hmc_m_scale3                2

hmc_dsf_k11                 2
hmc_dsf_k12                 3
hmc_dsfr_k1                 1
hmc_dsd                     2
hmc_dsig                    3
hmc_dsg                     4
```

The trajectory length is $\tau = 1$, with a total of 4 different integration time-scales with the second order minimal norm integrator `2MNSTS` on each scale; see `hmc_integrator` for other options. There are 5 integration steps on scale number 1 `hmc_steps`, two steps nested within this for scale number 2 `hmc_m_scale`, then nested with two steps again for

scales 3 then 4. Instead of nested integrators, one can use generalized integrators where each time-scale's integration scheme is independent: see section 5.4 for more information.

The time-scale numberings are referenced by the time scale specifiers which follow. The setup of this file leads to the following distribution of action terms on each time scale:

1. The single-fermion pseudofermion term (hmc_dsfr_k1)
2. The heavier Hasenbusch term (hmc_dsf_k11) and the Clover determinant (hmc_dsd)
3. The Hasenbusch correction term (hmc_dsf_k12) and the improved part of the gluon action (hmc_dsig)
4. The plaquette/Wilson part of the gluon action (hmc_dsg)

3.1.4. Markov Chain

Next are some parameters setting up the Markov chain

```
hmc_accept_first              10

start_configuration           hot
start_random                  319503
mc_steps                      10
mc_total_steps                20000
mc_save_frequency             0
```

This causes a hot start for the gauge field (start_configuration) with random seed '319503' (start_random). 10 trajectories are calculated during each run of the code (mc_steps) with the first 10 undergoing forced acceptance (hmc_accept_first). No configurations are saved to file (mc_save_frequency), and the code stops after 20,000 trajectories (mc_total_steps). The default for mc_total_steps is low, and should usually be set.

3.1.5. Fermion matrix inversion

Solver parameters:

```
solver_rest           1e-11
solver_rest_md        1e-9
solver_rest_cg_ritz   1e-11
solver_maxiter        1200
...
solver_outer_solver   cg
...
```

The matrix inverter used is CG, which is set by `solver_outer_solver`. The tolerance of this solver on fermion action evaluation is `1e-11` (`solver_rest`), whereas it is slightly less precise at `1e-9` during the molecular dynamics trajectories (`solver_rest_md`). Eigenvalues of the matrix are evaluated at the higher precision though (`solver_rest_cg_ritz`).

The maximum number of iterations is 1200 (`solver_maxiter`). This parameter has a default value of 100, and hence should be set explicitly.

See the input keyword documentation for information on the other solver keywords in section 3.3.19.

3.1.6. Running BQCD

Now to actually run BQCD! Copy `bqcd.300.input` to a working directory, change to that directory, then run the command

```
bqcd bqcd.300.input bqcd.300.output &> bqcd.300.1.log
```

where `bqcd` is a full path to the BQCD executable. This will take about 10 minutes to complete, and it produces the output file `bqcd.300.output`. Note that this command also redirects the standard output and error to `bqcd.300.1.log` — normally, this code would be run on a supercomputer, and these output streams would consequently go to their own files automatically.

The code also produces a LIME file `bqcd.300.lime`, which contains the gauge field configuration at the end of the run in ILDG format along with metadata for restarting the Markov chain. We can proceed to resume the Markov chain by running the code again with the `-c` flag. But first, we can speed things up by running the code in parallel: change the line with keyword `processes` to

```
processes  1 1 2 2
```

and now run the code over 4 processes

```
mpirun -np 4 bqcd -c bqcd.300.input bqcd.300.output &> bqcd.300.2.log
```

This will produce 10 more trajectories.

3.1.7. BQCD output

The output file generated by BQCD, `bqcd.300.output`, is broken down into sections by `>Begin/>End` delimiters. The output for each run is surrounded by `>BeginJob/>EndJob` tokens for ease of access.

The header section `>BeginHeader/>EndHeader` for each run contains information about when the job was started, and some of the input parameters.

The main part of the output file contains a series of embedded tables. These tables contain special tokens of the form `T%XX` and `%XX` such that we can use `grep` to extract them.

We can extract general information about the Markov Chain progress by running

```
grep '%mc' bqcd.300.output
```

This will give information about the plaquette, acceptance rate and iterations counts for each trajectory:

T%mc	traj	e	f	PlaqEnergy	exp(-Delta_H)	Acc	CGcalls	CGitTot	CGitMax	CGMcalls	CGMitTot	CGMitMax	Plaquette
%mc	1	1	1	0.4603695377	0.9406814248	1	53	2422	61	0	0	0	0.539630462296880
%mc	2	1	1	0.4597443535	0.9182754364	1	53	2290	55	0	0	0	0.540255646459386
%mc	3	1	1	0.4608069194	1.0295291357	1	53	2308	57	0	0	0	0.539193080649400
%mc	4	1	1	0.4591734179	1.0057202626	1	53	2206	55	0	0	0	0.540826582090360
%mc	5	1	1	0.4744552445	0.9244531373	1	53	1985	51	0	0	0	0.525544755494560
%mc	6	1	1	0.4717006653	0.9892924489	1	53	2011	47	0	0	0	0.528299334696576
%mc	7	1	1	0.4612058658	0.9370571005	1	53	2039	52	0	0	0	0.538794134178326
%mc	8	1	1	0.4527961003	0.9920165839	1	53	1989	50	0	0	0	0.547203899705222
%mc	9	1	1	0.4516703478	0.9742350535	1	53	2064	49	0	0	0	0.548329652150200
%mc	10	1	1	0.4544061729	0.9963916062	1	53	2105	51	0	0	0	0.545593827098611
T%mc	traj	e	f	PlaqEnergy	exp(-Delta_H)	Acc	CGcalls	CGitTot	CGitMax	CGMcalls	CGMitTot	CGMitMax	Plaquette
%mc	11	1	1	0.4609078806	1.0152287442	1	53	2067	52	0	0	0	0.539092119360002
%mc	12	1	1	0.4563820740	0.9691243600	1	53	2114	49	0	0	0	0.543617925997414
%mc	13	1	1	0.4684701708	0.9935836158	1	53	2039	53	0	0	0	0.531529829236492
%mc	14	1	1	0.4651568757	1.0140884160	1	53	2019	49	0	0	0	0.534843124331008
%mc	15	1	1	0.4502802111	0.9663791611	1	53	2021	50	0	0	0	0.549719788863034
%mc	16	1	1	0.4505187147	1.0078377885	1	53	1995	49	0	0	0	0.549481285258094
%mc	17	1	1	0.4468129429	0.9965722228	1	53	2023	50	0	0	0	0.553187057109728
%mc	18	1	1	0.4473054753	0.9839808214	1	53	2033	50	0	0	0	0.552694524687057
%mc	19	1	1	0.4452533591	1.0368457399	1	53	2116	50	0	0	0	0.554746640879882
%mc	20	1	1	0.4518358652	0.9818649052	1	53	2175	54	0	0	0	0.548164134753665
%mc	21	1	1	0.4637735116	0.9187633143	1	53	2126	53	0	0	0	0.536226488442716
%mc	22	1	1	0.4719595030	0.9990389368	1	53	2123	50	0	0	0	0.528040497005646
%mc	23	1	1	0.4711059887	1.0049051498	1	53	2262	53	0	0	0	0.528894011297565
%mc	24	1	1	0.4630871801	0.9687888230	1	53	2176	53	0	0	0	0.536912819873419
%mc	25	1	1	0.4773637154	0.9786564332	1	53	2228	52	0	0	0	0.522636284633002
%mc	26	1	1	0.4818968039	0.9527153539	1	53	2321	56	0	0	0	0.518103196065121
%mc	27	1	1	0.4849282090	0.9596876019	1	53	2287	57	0	0	0	0.515071791018083
%mc	28	1	1	0.4736714060	1.0016959817	1	53	2210	54	0	0	0	0.526328593955653
%mc	29	1	1	0.4736714060	0.9708984380	0	53	2235	54	0	0	0	0.526328593955653
%mc	30	1	1	0.4662033473	0.9745797939	1	53	2074	52	0	0	0	0.533796652740701

For the first 10 trajectories which were force-accepted, this data is contained in the table marked `%fa` instead.

We can also look at an iteration count breakdown in the table `%it`, and the average `%Favg` and maximal `%Fmax` forces. A comprehensive list of the embedded tables can be found in section 3.7.7.

A footer section `>BeginFooter/>EndFooter` for each run has the completion time of the job and the total CPU-Time.

At the end of each run is a timing section `>BeginTiming/>EndTiming`, which contains runtime statistics on many different operations within the code.

3.2. Command line

BQCD takes the following arguments:

```
bqcd [-c] input [output]
bqcd --convert-to-ildg input [output]
bqcd -I
bqcd -V
```

3.2.1. Flag `-c` (continuation job)

If the `-c` parameter is present the start configuration is being read from file. Otherwise a start configuration is being generated. The setup is such that `input` does not have to be modified from the first to the second job in a job chain.

3.2.2. Flag `--convert-to-ildg`

This flag is used to convert saved configuration from BQCD formats to the ILDG format (see section 6.7). Example:

- work with `bqcd.200.input` and modify `io_conf_format` (i.e. switch from `ildg` to `bqcd2` format)

  ```
  io_conf_format "bqcd2"
  ```

- run BQCD

  ```
  bqcd bqcd.200.input bqcd.200.output &> bqcd.200.3.log
  ```

- convert the gauge field configuration of trajectory 10 to ILDG format

  ```
  bqcd --convert-to-ildg bqcd.200.00010.info bqcd.200.00010.output
  ```

3.2.3. Flag `-I` (print default input values)

If `-I` is given the program prints all possible input parameters with their default values and exits. Most of them are described in section 3.3.

3.2.4. Flag `-V` (print program version)

The program prints version information and exits. The output looks like this:

```
This is bqcd 5.1 (revision 887)
   input format:               5
   conf info format:           4
```

```
MAX_TEMPER:                50
REAL kind:                  8
Version of D:             100
Communication:    MPI (sc:immediate) (g:immediate) + OpenMP
RandomNumbers:    ranlux-3.2 level 2
```

3.2.5. Argument `input`

Name of input parameter file.

3.2.6. Argument `output`

Name of log- and results file. If not given data will be written to *stdout*. If the file does not exist it will be created. If `-c` is set new output will be appended to the file. If `-c` is not set an existing file will be overwritten.

3.3. Input parameters and the syntax of `input` file

The syntax of the input file is one *keyword value(s)* pair (or tuple) per line. Empty lines and lines beginning with a `#` character are ignored. Keywords are checked for validity but the number of values and the types of values are not. It is a good idea to enclose character string parameters in double quotes `"..."` (in particular Fortran might scramble filenames containing slashes).

The following input keyword documentation is fairly self-explanatory but some keyword types are worth noting. An *enum* is a string which takes certain built-in values and a *flag* is an integer which switches a feature on and off, usually with off `= 0` and on otherwise.

3.3.1. General parameters

`run`

Type: integer; *Default:* `0`

An integer that specifies the run number of the job. This is stored in generated configurations, such that restarts are performed correctly. Try to give each distinct input file a different run number.

`comment`

Type: string; *Default:* `""`

A comment, which is (only) added to the header of the output. Useful to identify what configuration of parameters you are using.

3.3.2. Lattice and domain decomposition

`lattice`

Type: four integers; *Default:* `4 4 4 4`

A set of 4 integers that specifies the extent of the lattice in the usual order, i.e. '`L_X L_Y L_Z L_T`'. Each extent must be an even number in order to accommodate even-odd preconditioning.

`processes`

Type: four integers; *Default:* `1 1 1 1`

A set of 4 integers that specifies how to split the lattice across the processors, '`N_X N_Y N_Z N_T`'. The total number of processors is given by the product. The number of processors in each direction must evenly divide the corresponding lattice extent. BQCD also requires that `L_X/N_X` is even, such that it can implement even-odd preconditioning.

`ddlattice`

Type: four integers; *Default:* `1 1 1 1`

Specify the domain decomposition in each direction for DD preconditioning. No DD takes place if set to default.

The program must be compiled with the `-DBQCD_DDHMC` macro flag for DD preconditioning to be available.

`process_mapping`

Type: four integers; *Default:* `1 2 3 4`

A permutation of '`1 2 3 4`' which specifies the order in which the process coordinates are mapped to process ranks, see appendix C.

`boundary_conditions_fermions`

Type: four integers; *Default:* `1 1 1 -1`

A set of 4 integers which are either '`1`' or '`-1`' that specify the boundary conditions for the fermion fields in each direction. `1` is periodic, `-1` is anti-periodic.

`boundary_sf`

Type: flag; *Default:* `0`

A logical switch that determines whether we use Schroedinger's boundary conditions, see section 4.3.

3.3.3. Gauge action

Refer to section 4.1 for further explanation of gauge actions.

`gauge_action`

Type: enum; *Default:* `WILSON`

A string that specifies which gauge action to use.

- `WILSON`: use the Wilson gauge action
- `TREE`: use the tree-level improved action
- `IWASAKI`: use the Iwasaki action

See section 4.1 for expressions of these actions.

`beta`

Type: float; *Default:* `0.0`

Defines β, the inverse coupling of the $SU(3)$ gauge.

`hmc_dsg`

Type: integer; *Default:* `0`

Specifies the time-scale on which the plaquette (Wilson) gluon action S_{plaq} is integrated.

`hmc_dsig`

Type: integer; *Default:* `0`

Specifies the time-scale on which the improved part of the gluon action $S_{\text{imp}} = S_g - S_{\text{plaq}}$ is integrated.

3.3.4. Fermion action

Refer to section 4.2 for further information on fermion actions. For κ values, see `nf2_kappa[1-6]` and `nf1_kappa[1-6]`.

`fermi_action`

Type: enum; *Default:* `NON`

A string that specifies which fermion action to use.

- `NON`: no fermions; just do gluodynamics
- `WILSON`: use the Wilson fermion action
- `CLOVER`: use the $\mathcal{O}(a)$-improved 'clover' fermion action
  Requires `csw`
- `SLW`: use the Wilson fermion action with stout links

Requires `alpha`, `n_stout`

- `SLIC`: same as `CLOVER`, but with stout links in the clover term
  Requires `csw`, `alpha`, `n_stout`
- `SLRC`: same as `CLOVER`, but with stout links in the Wilson term
  Requires `csw`, `alpha`, `n_stout`
- `SLOC`: same as `CLOVER`, but with stout links in both terms
  Requires `csw`, `alpha`, `n_stout`

See section 4.2 for expressions of these actions.

`h`

Type: float; *Default:* `0.0`

Defines h, the twisted mass parameter for explicit parity-flavour symmetry breaking. See section 4.2 for the corresponding action term.

`chemi`

Type: float; *Default:* `0.0`

Defines the real part of the chemical potential μ. See section 4.2 for an explanation.

`chemi_i`

Type: float; *Default:* `0.0`

Defines the imaginary part of the chemical potential μ. See section 4.2 for an explanation.

`breaking_term`

Type: file; *Default:* `""`

Specifies the path to the matrix formed file to violate CPT symmetries with λ (`nf2_lambda[1-6]` and `nf1_lambda[1-6]`), see section 4.2.

`n_stout`

Type: integer; *Default:* `0`

Required for: a stout-link fermion action

Determines how many times to smear the gauge links for a stout link fermion action. See section 4.2 for an explanation.

`alpha`

Type: float; *Default:* `0.0`

Required for: a stout-link fermion action

Sets the parameter α for stout link smearing. See section 4.2 for an explanation.

`csw`

Type: float; *Default:* `0.0`

Required for: a clover fermion action

Defines c_{SW}, the Symanzik improvement coefficient for clover actions.

`hmc_dsd`

Type: integer; *Default:* `0`

Required for: a clover fermion action

Specifies the time-scale on which the clover determinant is integrated.

3.3.5. Double-flavour pseudofermions

`nf2_kappa[1-6]`

Type: float; *Default:* `0.0`

Specifies κ for up to 6 distinct double-flavour pseudofermions.

`nf2_kappa[1-6]h[1-3]`

Type: float; *Default:* `0.0`

Specifies up to three Hasenbusch masses for each double-flavour pseudofermion. These are ordered finest to coarsest, which is *reverse* to the usual action ordering: e.g. `nf2_kappa1h1` is heavier than `nf2_kappa1h2`. See the section 5.1 for more information.

`hmc_hkappa`

Type: flag; *Default:* `0`

Toggles the way Hasenbusch masses are implemented from the given values `nf2_kappa[1-6]h[1-3]`.

- On (non-zero): given values are modified κ parameters κ', such that $W = K(\kappa')$
- Off: given values are ρ shifts, such that $W = K(\kappa) + \rho$.

`nf2_lambda[1-6]`

Type: float; *Default:* `0.0`

Specifies λ for up to 6 distinct double-flavour pseudofermions to violate CPT symmetries with `breaking_term`, see section 4.2.

`hmc_dsf_k[1-6][1-4]`

Type: integer; *Default:* `0`

Specifies the time-scale on which the j^{th} mass-preconditioned term for the

i^{th} two-flavour fermion is integrated.

The ordering of the action terms is the *reverse* of the action term ordering; see section 3.3.18 for more information.

3.3.6. Single-flavour pseudofermions

`nf1_kappa[1-6]`

Type: float; *Default:* `0.0`

Specifies κ for up to 6 distinct single-flavour fermions.

`nf1_kappa[1-6]npf`

Type: integer; *Default:* `1`

Specifies how many degenerate pseudo-fermions there are for each specified single-flavour fermion.

`nf1_kappa[1-6]nth`

Type: integer; *Default:* `2`

Specifies the n-th root of $K = M^\dagger M$ to be approximated by the RHMC rational approximation for each single-flavour fermion.

`nf1_lambda[1-6]`

Type: float; *Default:* `0.0`

Specifies λ for up to 6 distinct single-flavour pseudofermions to violate CPT symmetries with `breaking_term`, see section 4.2.

`hmc_dsfr_k[1-6]`

Type: integer; *Default:* `0`

Specifies the time-scale on which the RHMC term is integrated for each single-flavour fermion. See time-scale section section 3.3.18 for more information.

`rescale_rhmc`

Type: flag; *Default:* `1`

Determines (non-zero = on) whether to rescale the RHMC approximation after each trajectory. If the ratapp has range $[l_{min}, l_{max}]$ and the fermion matrix has eigenvalue range $[\lambda_{min}, \lambda_{max}]$, we can use

$$R(K) \approx K^n = a^{-n}(aK)^n \approx a^{-n}R(aK) \tag{1}$$

to centre the approximation's effective range on the true eigenvalue spectrum.

Namely, $R(aK)$ is effective on $[l_{min}/a, l_{max}/a]$, so we choose a such that

$$\frac{l_{max}}{a\lambda_{max}} = \frac{a\lambda_{min}}{l_{min}}$$
$$\implies \qquad a = \sqrt{\frac{l_{min}l_{max}}{\lambda_{min}\lambda_{max}}}$$

Turning this off is useful when comparing methods, as it ensures that the rational approximation does not change.

3.3.7. RHMC tuning parameters

`tuning_approx_range`

Type: flag; *Default:* `0`

Turns on tuning of rational approximation.

`tuning_approx_range_list`

Type: file; *Default:* `""`

Specifies file to tuning rational approximation, see data/rangelist in the source code for an example.

`tuning_fraction_tolerance`

Type: file; *Default:* `""`

Specifies file to tuning fraction tolerances, see data/fractiontolerance in the source code for an example.

3.3.8. Zolotarev rational approximation

For an brief explanation of the Zolotarev approximation, see section 5.6.

`hmc_zolo`

Type: flag; *Default:* `0`

A switch that determines whether the program uses a Zolotarev rational approximation for RHMC rather than the built-in Remez algorithm for the case where `nf1_kappa[1-6]nth=2`. It only takes effect for such pseudofermions because the Zolotarev rational approximation is an approximation to the inverse square root.

`hmc_zolo_n`

Type: integer; *Default:* `0`

Specifies the number of shifts n to use in the Zolotarev approximation. If omitted, the number of shifts is calculated to produce the error delta

hmc_zolo_delta on the given approximation range hmc_zolo_[min|max].

`hmc_zolo_delta`

Type: float; *Default:* `1e-6`

Specifies the error delta for the Zolotarev approximation, which is the maximum error of the approximation over the eigenvalue range specified by hmc_zolo_[min|max].

If both `hmc_zolo_delta` and hmc_zolo_n are given, the latter takes preference, and the code will issue a warning if the produced rational approximation has a larger error delta.

`hmc_zolo_[min|max]`

Type: integer; *Default:* `0`

Specifies the minimum/maximum eigenvalue bound to approximate in the Zolotarev approximation. If omitted, the eigenvalue range is taken to be $[0.8\lambda_{min}, 1.2\lambda_{max}]$, where $\lambda_{min}/\lambda_{max}$ is the calculated minimum/maximum eigenvalue for the fermion matrix $M^\dagger M$.

3.3.9. QED

For more information on using QED, see section 4.4.

`beta_qed`

Type: float; *Default:* `0.0`

Defines β_{QED}, the inverse coupling of the electromagnetic force.

`nf2_em_charge[1-6]_x3`

Type: integer; *Default:* `0`

Defines how many thirds of the electron charge the i^{th} double-flavour pseudofermion has.

`nf1_em_charge[1-6]_x3`

Type: integer; *Default:* `0`

Defines how many thirds of the electron charge the i^{th} single-flavour pseudofermion has.

3.3.10. Axion

`kappa_axion`

Type: float; *Default:* `0.0`

Defines κ_a, the hopping parameter of axion. See section 4.5.

`finv_axion`

Type: float; *Default:* `0.0`

Defines f_{inv}, a relevant parameter of the inverse decay constant of axion. See section 4.5.

3.3.11. PFHMC

For an explanation of polynomial filtering, including the formulation of the polynomials used, see section 5.3.

`nf2_k[1-6]p[1-3]`

Type: integer; *Default:* `0`

Specifies up to 3 polynomial filters via their order for each possible two-flavour degenerate pseudo-fermion. They are expected to be ordered from finest to coarsest, meaning that (for example) `nf2_k1p1` > `nf2_k1p2`. The given polynomial orders are passed to the Chebyshev polynomial creation routines to create polynomials of suitable order and (μ, ν).

`nf2_k[1-6]_[mu|nu]`

Type: float; *Default:* `0.0`

Specifies the value of the μ/ν parameter for the Chebyshev polynomial used in PFHMC for each double-flavour pseudofermion. Together, these two parameters specify the ellipse upon which the roots of the Chebyshev polynomials lay: see section 5.3 for further explanation.

`nf2_k[1-6]p[1-3]_file`

Type: file; *Default:* `""`

Specifies up to three polynomial filters for each possible double-flavour pseudofermion which are loaded from the specified file. This file should contain the following

- Line 1: the polynomial order n
- Lines 2 to $n+1$: the polynomial roots z_i, specified as complex numbers via 2-tuples `(x,y)` $= x + iy$.

  IMPORTANT: the current implementation of polynomial filtering assumes that $z_i^* = z_{n-i+1}$, so the roots should be specified in an order that obeys this.

See `data/testpoly_nf2_p4.txt` for an example.

The polynomials are expected to be defined from finest to coarsest, i.e. in

decreasing order.

The polynomial is loaded directly into the action term, giving $S_i = \phi_i^\dagger P(K)\phi_i$. This is important to note when using multiple filters. The polynomial for the correction term $\phi^\dagger[KP(K)]^{-1}\phi$ is calculated in the code as the product of all the polynomial filters $P(K)$ plus a zero root K.

`hmc_dsfp_k[1-6][1-4]`

Type: integer; *Default:* `0`

Specifies the integration time-scales for each part of the specified polynomial-filtered actions.

Note that due to the way the polynomial filters are initialized, the action terms are in the *reverse* order. See section 3.3.18 for more details.

`nf1_k[1-6]p1`

Type: integer; *Default:* `0`

Specifies a polynomial filter via its order for each single-flavour pseudo-fermion. This polynomial is a built-in Chebyshev polynomial that approximates the inverse square root over the range `[5e-3,3]`.

For more polynomial filters in PF-RHMC, please use the polynomial reading facilities provided by `nf1_k1p1_file`.

`nf1_k[1-6]p[1-3]_file`

Type: file; *Default:* `""`

Specifies up to three polynomial filters for each possible single-flavour pseudo-fermion via loading from the specified file. They are expected to be defined from finest to coarsest.

See `nf1_k[1-6]p[1-3]_file` for an explanation of the file format. An example file is available at `data/testpoly_nf1_p4.txt`.

The polynomial for the correction term $\phi^\dagger P(K)^{-1}R(K)\phi$ is calculated in the code as the product of all the polynomial filters.

`hmc_dsfr_k[1-6]p[1-3]`

Type: integer; *Default:* `0`

Specifies the integration time-scales for each part of the specified polynomial-filtered actions for single-flavour pseudo-fermions.

Note that due to the way the polynomial filters are initialized, the action terms are in the *reverse* order. See section 3.3.18 for more details.

3.3.12. Truncated rational HMC (tRHMC)

For an explanation of truncated RHMC, see section 5.5. Note that the current implementation requires the use of the Zolotarev approximation (`hmc_zolo`).

`nf1_k[1-6]_trunc[1-3]`

Type: integer; *Default:* `0`

Specifies up to 3 truncation indices for each single-flavour rational approximation RHMC. The specified indices are the points at which the rational approximation is cut, with the ratapp shifts being ordered from largest `t=1` to smallest `t=n`.

As the code assumes filters are ordered from finest to coarsest, the filters should be decreasing.

`hmc_dsfr_k[1-6]t[1-3]`

Type: integer; *Default:* `0`

Specifies the integration time-scales for tRHMC actions. These are ordered in *reverse* relative to the action terms, see section 3.3.18 for more details.

3.3.13. Start parameters

The following parameters only take effect if the `-c` flag is not used, implying that the job is an initial HMC run.

`start_configuration`

Type: enum; *Default:* `cold`

A string that determines how the initial gauge configuration is produced:

- `cold`: start from scratch with $U = I$
- `hot`: start from scratch with U set randomly
- `file`: load from a given file. Requires `start_ildg_file` xor `start_info_file`.

`start_ildg_file`

Type: file; *Default:* `""`

Specifies the ILDG `.lime` file from which we should read the initial gauge configuration. Mutually exclusive with `start_info_file`.

`start_info_file`

Type: file; *Default:* `""`

Specifies the BQCD `.info` file from which we should read the initial gauge configuration. Mutually exclusive with `start_ildg_file`.

`start_random`

Type: enum or integer; *Default:* `""`

Specifies the random seed (an integer) that initializes the random number generator at the start of a Markov chain:

- `default`: use a default seed built into BQCD.
- `random`: produce a seed at random.
- (integer): use the given integer as the seed.

Of these three options, the third is recommended as this ensures that the runs are reproducible yet still 'random' between different runs.

3.3.14. Configuration I/O

For further information, see section 3.4 to section 3.6.

`io_restart_format`

Type: enum; *Default:* `bqcd`

Specify the format `[bqcd|bqcd2|ildg]` for reading/saving configurations at the start/end of each execution for checkpointing. Such configurations are saved at the end of every execution of the code, and read when starting a continuation run via `-c` (see also section 6.7).

`io_conf_format`

Type: enum; *Default:* `bqcd`

Specify the format `[bqcd|bqcd2|ildg]` for saving configurations after certain trajectories as specified by `mc_save_frequency` (see also section 6.7).

`io_bqcd_restart_filename`

Type: string; *Default:* `"bqcd.%R3"`

Specify the names (w/o extension) of checkpoint/restart files in `bqcd` or `bqcd2` `io_restart_format` (for the `%`-macro expansion see section 3.5).

`io_bqcd_conf_filename`

Type: string; *Default:* `"bqcd.%R3.%T5"`

Specify the names (w/o extension) of files for saved configurations in `bqcd` or `bqcd2` `io_conf_format` (for the `%`-macro expansion see section 3.5).

`ildg_precision`

Type: integer; *Default:* `64`

Sets the precision (in bits `[32|64]`) for ILDG files saved explicitly via `mc_save_frequency`.

`ildg_precision_restart`

Type: integer; *Default:* `64`

Sets the precision (in bits `[32|64]`) for checkpoint ILDG files.

`ildg_filename_prefix`

Type: string; *Default:* `"bqcd.%R3"`

Specifies the prefix used when explicitly outputting ILDG files. This should contain information about the simulated ensemble as a whole. (for the %-macro expansion see section 3.5)

Saved ILDG configurations are named `<prefix><middle>.<extension>`, and ILDG restart files are named `bqcd.<run>.<extension>`, where

- `<prefix>` = `ildg_filename_prefix`
- `<middle>` = `ildg_filename_middle`
- `<extension>` = `ildg_filename_extension`
- `<run>` = `run`

`ildg_filename_middle`

Type: string; *Default:* `".%T5"`

Specifies the 'middle' part used when explicitly outputting ILDG files. This should about contain information each individual configuration. (for the %-macro expansion see section 3.5)

`ildg_filename_extension`

Type: string; *Default:* `"lime"`

Specifies the extension part used when outputting ILDG files.

`ildg_template_ensemble`

Type: file; *Default:* `""`

Specifies the path to the template XML file for saving ensemble metadata. The produced ensemble XML file is saved as `<prefix>.xml` (see `ildg_filename_prefix` for an explanation).

`ildg_template_conf`

Type: file; *Default:* `""`

Specifies the path to the template XML file for saving per-configuration metadata. The produced configuration XML files are saved as `<prefix><middle>.xml` (see `ildg_filename_prefix` for an explanation).

`ildg_markov_chain_uri`

Type: string; *Default:* `"mc://UNDEFINED"`

Specifies the Markov Chain URI to be placed in the ensemble XML file. It fills out the corresponding template parameter in the XML file. This information is used for tagging on an ILDG database.

`ildg_data_lfn_path`

Type: string; *Default:* `"lfn://UNDEFINED"`

Specifies the LFN for the ensemble XML file. It fills out the corresponding template parameter in the XML file. This information is used for tagging on an ILDG database.

`ildg_participant_name`

Type: string; *Default:* `""`

Field for the code user's name in the XML files. It fills out the corresponding template parameter in the XML file.

`ildg_participant_institution`

Type: string; *Default:* `""`

Field for the code user's institution in the XML files. It fills out the corresponding template parameter in the XML file.

`ildg_machine_name`

Type: string; *Default:* `""`

Field for the name of the machine BQCD is used on in the XML files. It fills out the corresponding template parameter in the XML file.

`ildg_machine_institution`

Type: string; *Default:* `""`

Field for the institution of the machine BQCD is used on in the XML files. It fills out the corresponding template parameter in the XML file.

`ildg_machine_type`

Type: string; *Default:* `""`

Field for the type of the machine BQCD is used on (e.g. BlueGene) in the XML files. It fills out the corresponding template parameter in the XML file.

3.3.15. Markov Chain

`mc_steps`

Type: integer; *Default:* `1`

An integer that specifies how many Markov Chain trajectories should be

calculated for this execution run.

`mc_save_frequency`

Type: integer; *Default:* `1`

An integer that specifies how often to save the gauge configuration. If `mc_save_frequency` $= n$, then every n^{th} gauge field configuration is saved to file. The file type is specified by `io_conf_format`.

`mc_total_steps`

Type: integer; *Default:* `1`

An integer that specifies the maximum number of Markov Chain trajectories that should be calculated over all runs from this input file. When the trajectory counter reaches this number, a `.STOP` file is produced and no more trajectories are calculated.

3.3.16. Hybrid Monte Carlo

`hmc_trajectory_length`

Type: integer; *Default:* `1`

Specifies the length τ of each trajectory.

`hmc_accept_first`

Type: integer; *Default:* `0`

An integer that specifies how many trajectories at the start of a simulation are forced to be accepted. This is necessary for some configurations to ensure we don't get stuck. The force acceptance can span several runs of the code. The logs for these trajectories are given in the table `%fa`.

This should be set to zero if starting from a file, as forced acceptance will inevitably take the system out of equilibrium.

`hmc_test`

Type: flag; *Default:* `0`

An integer that acts as a logical flag for HMC reversibility testing. This causes BQCD to calculate a trajectory both forwards and backwards, report on the difference between the initial and final states in the section `HMCtest` of the output file, then finish.

3.3.17. Integrator specification

`hmc_steps`

Type: integer; *Default:* `0`

An integer that specifies how many integration steps to take per trajectory at the coarsest scale, time-scale #1. The corresponding step-size h is given by $h = \frac{\tau}{n_{\text{steps}}}$.

`hmc_genint`

Type: flag; *Default:* `0`

A flag for activating the use of the generalized multi-scale integration scheme, as opposed to the default nested multi-scale scheme. This scheme overlays different integration schemes for each time-scale onto a single time step axis, such that any number of steps can be used on each time-scale. See section 5.4 for more details.

Note that using the generalized scheme changes the meaning of `hmc_m_scale`.

`hmc_m_scale, hmc_m_scale[2-5]`

Type: integer; *Default:* `1`

An integer that specifies, by default, the relative scaling between successive time-scales. If `hmc_genint` is on, then this keyword specifies the absolute number of integration steps at this time-scale.

In the default case, if `hmc_m_scale=2`, then for every space update step at the top level we have two integration steps at the second level. Note that this does not necessarily mean there are 2 space steps on the second level for every space step in the top level – an integration step can have several space steps.

The time scales are indexed as `1:hmc_steps`, `2:hmc_m_scale`, `3:hmc_m_scale2` etc. This indexing is used by both the time-scale specification keywords (section 3.3.18), and the integrator specification keywords `hmc_integrator`.

`hmc_integrator[1-6]`

Type: enum; *Default:* `LPFSTS` for 1–3, `NON` for 4–6

Strings that specify which integrators to use at each time-scale. and *must* be specified for each time-scale used. The number of integration steps at each time-scale are defined via `hmc_steps` and `hmc_m_scale`.

- `LPFSTS`: use the space-time-space leapfrog integrator
- `LPFTST`: use the time-space-time leapfrog integrator
- `2MNSTS`: use the space-time-space 2^{nd} order minimal-norm integrator
- `2MNTST`: use the time-space-time 2^{nd} order minimal-norm integrator
- `4MN4FP`: use the position-space version of the 4^{th} order minimal-norm integrator

- `4MN5FV`: use the vector-space version of the 4^{th} order minimal-norm integrator
- `NON` and otherwise: treated as a blank; will cause an error if this scale is required.

3.3.18. Time scale specifiers

Also see `hmc_m_scale`, `hmc_integrator`.

- `hmc_dsg`: The integration time-scale for the plaquette part of the gluonic action, S_{plaq}.
- `hmc_dsig`: The integration time-scale for the improved part of the gluonic action, $S_g - S_{\text{plaq}}$.
- `hmc_dsd`: The integration time-scale for the clover determinant, S_{det}.
- `hmc_dsf_k[1-6][1-4]`: The integration time-scales for the double-flavour pseudofermions for standard HMC and mass preconditioned HMC.

  The corresponding action terms go from *finest to coarsest*, which is reverse from the usual action decomposition. For example, for a single Hasenbusch filter

  $$S_F = \phi_1^\dagger (W^\dagger W)^{-1} \phi_1 + \phi_2^\dagger W (M^\dagger M)^{-1} W^\dagger \phi_2, \tag{2}$$

  `hmc_dsf_k11` specifies the scale on which S_2 (the fine, light correction term) is integrated and `hmc_dsf_k12` specifies the scale on which S_1 (the coarse, heavy filter term) is integrated.
- `hmc_dsfp_k[1-6][1-3]`: The integration time-scales for the double-flavour pseudofermions for polynomial-filtered HMC.

  The corresponding action terms go from *finest to coarsest*, which is reverse from the usual action decomposition. Also note that in a pure polynomial-filtered case, the keywords `hmc_dsf_k[1-6]1` specify the scale of the final correction term.

  For example, in the two filter case, the low order polynomial filter term $S_1 = \phi_1^\dagger P_1(K) \phi_1$ has order $p_1 =$ `hmc_k1p2` and time scale `hmc_dsfp_k12`, the intermediate term $S_2 = \phi_2^\dagger Q(K) \phi_2$ has order $q = p_2 - p_1$ where $p_2 =$ `hmc_k1p1` and time scale `hmc_dsfp_k11`, and the correction term $S_3 = \phi_3^\dagger [P_2(K) K]^{-1} \phi_3$ has time scale `hmc_dsf_k11`.

  In the case of polynomial-filtered Hasenbusch (PF-MP), we have polynomial filters on top of Hasenbusch filters, so the polynomial correction term has time scale `hmc_dsf_k12` corresponding to the heaviest Hasenbusch term.
- `hmc_dsfr_k[1-6]`: The integration time-scales for the single-flavour pseudofermions with RHMC.

- `hmc_dsfr_k[1-6]p[1-3]`: The integration time-scales for the single-flavour pseudofermions for polynomial-filtered RHMC. The same ordering caveats apply as for `hmc_dsfp_k[1-6][1-3]`.
- `hmc_dsfr_k[1-6]t[1-3]`: The integration time-scales for the single-flavour pseudofermions for truncated RHMC.

  The corresponding action terms go from *finest to coarsest*. For example, in the double truncation case, the cheapest truncation term S_1 has time scale `hmc_dsfr_k1t2` and has a rational approximation that spans indices `[1, nf1_k1_trunc2]`, the intermediate term S_2 has time scale `hmc_dsfr_k1t1` and spans `[nf1_k1_trunc2+1, nf1_k1_trunc1]`, and the final correction term S_3 has time-scale `hmc_dsfr_k1` and spans `[nf1_k1_trunc1+1, n]`.

3.3.19. Solver parameters

`solver_outer_solver`

Type: enum; *Default:* `cg`

Specifies the iterative solver used to invert fermion matrix $W^\dagger W$.

- `cg`: standard conjugate gradient
- `dd`: domain decomposition solver
- `bicgstab`: stabilized bi-conjugate gradient
- `gmres`: generalized minimal residual method
- `gcrodr`: generalized conjugate residual with inner orthogonalization with deflated restarting
- `cg_mix`: mixed-precision conjugate gradient
- `bicgstab_mix`: mixed-precision stabilized bi-conjugate gradient
- `qudacg`: conjugate gradient by using QUDA

`solver_inner_solver`

Type: enum; *Default:* `cg`

Specifies the inner iterative solver for the mixed-precision inversion schemes.

- `cg`: standard conjugate gradient
- `bicgstab`: stabilized bi-conjugate gradient

`solver_outer_steps`

Type: integer; *Default:* `20`

For mixed precision inversion schemes, specifies how many times we use the outer loop to refine the solution.

`solver_rest`

Type: float; *Default:* `1e-8`

Specifies the solver tolerance when inverting the fermion matrix outside of the molecular dynamics trajectories.

`solver_rest_md`

Type: float; *Default:* `1e-8`

Specifies the solver tolerance when inverting the fermion matrix during molecular dynamics trajectories.

`solver_maxiter`

Type: integer; *Default:* `100`

Specifies the maximum number of solver iterations used to invert the fermion matrix before the program prints an error message and halts, unless using `solver_ignore_no_convergence`.

`solver_ignore_no_convergence`

Type: flag; *Default:* `0`

A logical flag that determines whether we should continue if we have reached `solver_maxiter`. This is usually a bad idea, so it needs to be set to `2` to be activated.

`solver_check_solution`

Type: flag; *Default:* `0`

A logical flag to determine whether to check the solutions to fermion matrix inversion explicitly. If switched on (non-zero), the program checks the solutions and outputs several new tables to the main output stream to show the progression of the solver. This adds to the compute time, and adds a *lot* of debug info to the output file, so only use if necessary.

`solver_mre_vectors`

Type: integer; *Default:* `0`

Specify how many previous inversion results to use as a basis for the next initial guess for a fermion matrix inversion via a minimum residual extrapolation (MRE). This can drastically improve the inversion speed. The paper on this method suggests using `7` for good results.

`solver_rest_cg_ritz`

Type: float; *Default:* `1e-8`

Specifies the solver tolerance when inverting the fermion matrix for calculating its eigenvalues.

`solver_stopping_criterion`

Type: enum; *Default:* `1`

Specifies which stopping criterion is used during fermion matrix inversion. Writing the system as $Ax = b$, the possible criterion are:

- `1`: $r^\dagger r <= \text{tol}$, where $r = Ax - b$ is the residue.
- `2`: $\sqrt{r^\dagger r} <= \sqrt{|b|} \times \text{tol}$.

`fullsolver`

Type: enum; *Default:* `eo`

Specifies the pre-conditioning used when inverting the fermion matrix W.

- `eo`: construct an even-odd preconditioned inverse with the iterative solver `mtilsolver`
- `dd`: use domain decomposition

`mtilsolver`

Type: enum; *Default:* `CGN`

Specifies the iterative solver used to invert W when `fullsolver=eo`.

- `CGN`: construct from the inverse of $W^\dagger W$, which is solved by conjugate gradient
- `bicgstab`: invert W via stabilized bi-conjugate gradient

`multi_shift_block_cg_qr`

Type: flag; *Default:* `0`

Switches multi shift CG to multi shift blocked CG with QR decomposition. Requires LAPACK.

3.3.20. Measurements

`measure_minmax`

Type: flag; *Default:* `0`

Turns on the measurement of the minimum and maximum eigenvalues for each Dirac matrix $W^\dagger W$ at each trajectory. This is performed using the CG-Ritz algorithm.

The results are stored in the table `%egnv` .

`measure_hadspec`

Type: file; *Default:* `""`

This keyword facilitates the construction of meson and baryon quark prop-

agators for each configuration during the 'measuring' phase of the process. The provided file determines what is measured and where the output of this process goes: see `data/myhpara` in the source code for an example.

`measure_only`

Type: flag; *Default:* `0`

If this flag is on, BQCD will skip the usual HMC process and only perform measurements on the saved configurations generated by BQCD. In this mode, `mc_steps` determines how many configurations are read then measured, starting from the trajectory `measure_start_traj`.

`measure_start_traj`

Type: integer; *Default:* `0`

Sets the trajectory number to start taking measurements from in `measure_only` mode.

`measure_cooling_list`

Type: file; *Default:* `""`

Specifies a file containing a list of cooling steps to compute topological charge, see section 3.8.1.

`measure_polyakov_loop`

Type: flag; *Default:* `0`

Turns on Polyakov loop measurement, see section 3.8.2.

`measure_traces`

Type: integer; *Default:* `0`

Turns on and specifies the number of noises to measure fermionic bulk quantities, see section 3.8.3.

`measure_traces_file`

Type: file; *Default:* `trace.log`

Specifies the log file for fermionic bulk quantities measurement, see section 3.8.3.

`measure_chemical`

Type: integer; *Default:* `0`

Specifies how often to measure the phase and log absolute value of the fermionic determinant by using the direct method, see section 3.8.4.

`measure_chemical_file`

Type: file; *Default:* `""`

Specifies the log file for measure_chemical.

`measure_schrpcac`

Type: integer; *Default:* `0`

Specifies how often to measure f_A and f_P, used to determine non-perturbative c_{SW} via the Schrödinger functional method; see section 3.8.5.

`measure_schrpcac_kappa`

Type: float; *Default:* `0`

Specifies κ when f_A and f_P are measured with measure_schrpcac.

`measure_schrpcac_csw`

Type: float; *Default:* `0`

Specifies c_{SW} when f_A and f_P are measured with measure_schrpcac.

`measure_schrpcac_lambda`

Type: float; *Default:* `0`

Specifies λ when f_A and f_P are measured with measure_schrpcac.

`measure_schrpcac_em_charge`

Type: float; *Default:* `0`

Specifies electromagnetic charge when f_A and f_P are measured with measure_schrpcac.

`measure_schrpcac_file`

Type: file; *Default:* `""`

Specifies the log file for measure_schrpcac.

`measure_wilson_flow`

Type: integer; *Default:* `0`

Turns on Wilson flow measurement for QCD SU(3) field, see section 3.8.6.

`measure_wilson_flow_file`

Type: file; *Default:* `""`

Specifies the log file for measure_wilson_flow.

`measure_wilson_flow_steps`

Type: integer; *Default:* `1000`

Specifies the number of Runge–Kutta steps for measure_wilson_flow.

`measure_wilson_flow_eps`

Type: float; *Default:* `1d-3`

Specifies the Runge–Kutta step size for measure_wilson_flow.

measure_wilson_flow_qed

Type: integer; *Default:* 0

Turns on Wilson flow measurement for QED U(1) field, see section 3.8.7.

measure_wilson_flow_qed_file

Type: file; *Default:* ""

Specifies the log file for measure_wilson_flow_qed.

measure_wilson_flow_qed_steps

Type: integer; *Default:* 1000

Specifies the number of Runge–Kutta steps for measure_wilson_flow_qed.

measure_wilson_flow_qed_eps

Type: float; *Default:* 1d-3

Specifies the Runge–Kutta step size for measure_wilson_flow_qed.

3.3.21. Compute performance tuning

tuning_cg_version

Type: integer; *Default:* 1

Selects the implementation of *conjugate gradient* solvers. Values: 1 or 2, see section 7.1.

tuning_cg_d

Type: integer; *Default:* 2

Selects the implementation of the hopping matrix multiplication when using tuning_cg_version=2. Values: 2, 21, 25 or 35; see section 7.2 and table 1).

tuning_cg_simd

Type: integer; *Default:* 0

If set to 1 the optimized SIMD implementation (see appendix D) is used if tuning_cg_version is 2.

tuning_cg_spincol

Type: integer; *Default:* 1

Selects the layout of spin-colour arrays if tuning_cg_version is 2. Values: 1 or 22 (see section 7.3.1).

tuning_cg_clover

Type: integer; *Default:* `1`

Selects the layout of clover arrays if `tuning_cg_version` is `2`. Values: `1` or `2` (see section 7.3.2).

`tuning_cg_loop_blocking`

Type: integer; *Default:* `0`

If set to `1` loop blocking (see appendix E) is used if `tuning_cg_version` is `1` (experimental). Has no meaning if `tuning_cg_version` is `2`.

`tuning_cg_block_length`

Type: integer; *Default:* `256`

Sets the block length for loop blocking (see appendix E) if `tuning_cg_loop_blocking` is `1`. Has no meaning if `tuning_cg_version` is `2`.

`tuning_cg_mcg_block_length`

Type: integer; *Default:* `0`

If set, defines the block length in loop blocking (see appendix E) for the multi-shift solver if `tuning_cg_version` is `2` (experimental).

`tuning_cg_precision_r4`

Type: integer; *Default:* `0`

If set to `1` *cg* is run in single precision if `tuning_cg_version` is `1` (experimental).

`tuning_cg_precision`

Type: integer; *Default:* `64`

If set to `32` *cg* is run in single precision if `tuning_cg_version` is `2` (experimental).

`tuning_xbound_sc_i`

Type: integer; *Default:* `1`

Sets the MPI communication pattern for spin-colour arrays. Values: `0` or `1`; see section 7.4.1.

`tuning_xbound_g_i`

Type: integer; *Default:* `1`

Sets the MPI communication pattern for gauge field arrays. Values: `0` or `1`; see section 7.4.1.

`solver_cg_poly`

Type: flag; *Default:* `0`

Specify which force calculation method to use for the polynomial correction

term in PF-HMC:

- `0` (default): Expand the inverse $[P(K)K]^{-1}$ as a sum over poles, then calculate the resultant terms using multi-shift CG to get the shifted inverses $[K + z_i]^{-1}$.
- `1`: Treat $P(K)K$ as a single matrix, and invert it with conjugate gradient and MRE vectors.
- `2`: Same as `0`, but invert $P(K)$ *then* K when constructing $[P(K)K]^{-1}\phi$.

3.3.22. Miscellaneous

`replay_trick_ntau`

Type: integer; *Default:* `0`

Specifies the number of steps on the coarsest scale (replacing `hmc_steps`) when we use the *replay trick*, and also acts as the flag for the replay trick.

The replay trick is simple: if a trajectory has change in the Hamiltonian `|dH| > replay_trick_threshold`, we do the HMC trajectory again with `replay_trick_ntau` steps at the coarsest scale. It also reports on the success of this technique: whether `|dH|` is reduced, is increased, or the new trajectory is not accepted.

`replay_trick_threshold`

Type: float; *Default:* `1.0`

Specifies the lower bound for the absolute change in the Hamiltonian `|dH|` to activate the replay trick. See `replay_trick_ntau` for details.

3.4. File naming conventions

By default file names have one these structures (see examples given below)

```
bqcd.run

bqcd.run.extension
bqcd.run.timeslice.extension

bqcd.run.trajectory.extension
bqcd.run.trajectory.timeslice.extension
```

where ***run*** is a three digit `run` number that is set in the input file, ***trajectory*** is a five digit trajectory counter. ***timeslice*** is a two digit time coordinate of a time slice (it will be extended automatically to three digits if $L_t > 99$).

extension is set by the program for bqcd and bqcd2 formats, for the ildg format it can be set by the ildg_filename_extension input parameter.

Output formats are described in section 6.7.

3.4.1. input, output and batch log files

These file names are not automatically being generated by the program. We choose names that fit to the naming scheme:

bqcd.200	input (command line parameter)
bqcd.200.res	output (command line parameter)
bqcd.200.input	input (command line parameter)
bqcd.200.output	output (command line parameter)
bqcd.200.log	log file from batch system

3.4.2. Restart files in bqcd format

bqcd.200.count	counters: run, job, trajectory
bqcd.200.info	configuration metadata
bqcd.200.ran	state of random number generator
bqcd.200.00.u	timeslice 0 of SU(3) configuration
bqcd.200.01.u	timeslice 1, ...
bqcd.200.02.u	
bqcd.200.03.u	

3.4.3. Restart files in bqcd2 format

bqcd.200.count	counters: run, job, trajectory
bqcd.200.info	configuration metadata
bqcd.200.ran	state of random number generator
bqcd.200.su3	SU(3) configuration

3.4.4. Restart files in lime format

bqcd.200.lime	all restart information

3.4.5. Configuration files in bqcd format

bqcd.200.00010.info	configuration metadata
bqcd.200.00010.00.u	configuration at trajectory 10, timeslice 0
bqcd.200.00010.01.u	configuration at trajectory 10, timeslice 1

```
bqcd.200.00010.02.u    ...
bqcd.200.00010.03.u
```

3.4.6. Configuration files in `bqcd2` format

`bqcd.200.00010.info`	configuration metadata
`bqcd.200.00010.su3`	configuration at trajectory 10

3.4.7. Configuration files in `ildg` format

`bqcd.200.xml`	ensemble metadata
`bqcd.200.00010.xml`	metadata of configuration at trajectory 10
`bqcd.200.00010.lime`	binary data of configuration at trajectory 10

3.5. Flexible filenames

The advantage of the default filenames described in section 3.4 is that the names are short. However, many users prefer to put characteristic metadata into filenames (e.g. β, κ and the lattice size).

The following macros can be used in filename specifications:

macro	replacement
%BE	beta
%KL	nf2_kappa1
%KS	nf1_kappa1
%LL	nf2_lambda1
%LS	nf1_lambda1
%LX	lattice(1)
%LY	lattice(2)
%LZ	lattice(3)
%LT	lattice(4)
%R*n*	run (*n* = 1..9 digits)
%T*n*	trajectory counter (*n* = 1..9 digits)

The macros can be used with these input parameters:

```
io_bqcd_restart_filename
io_bqcd_conf_filename
ildg_filename_prefix
ildg_filename_middle
```

3.6. Working with data in ILDG format

3.6.1. Restart files

By default the program works with restart files in its own `bqcd` format. To work with restart files in `ildg` format one has to set

```
io_restart_format ildg
```

in the input parameter file.

3.6.2. SU(3) configuration files and metadata

In order to work with the `ildg` data format one has to set:

```
io_conf_format ildg
```

The program will then write binary data in *lime* format as well as ensemble and configuration metadata. Currently *lime* I/O is sequential (the `bqcd` format allows for parallel I/O).

Generation of metadata works with templates. On has to provide template files containing placeholders. The syntax for placeholders is `#placeholder#`, for example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<gaugeConfiguration xmlns="http://www.lqcd.org/ildg/QCDml/config1.3">
    <management>
        <crcCheckSum>#crc_check_sum#</crcCheckSum>
        <archiveHistory>
            <elem>
                <revisionAction>generate</revisionAction>
                <participant>
                    <name>#participant_name#</name>
                    <institution>#participant_institution#</institution>
                </participant>
                <date>#today#</date>
            </elem>
        </archiveHistory>
    </management>
    ...
```

The following placeholders are available:

```
#participant_name#
```

```
#participant_institution#
#machine_name#
#machine_institution#
#machine_type#
#code_name#
#code_version#
#code_date#
#para_number_steps#
#para_step_size#
#para_time_scale_ratio#
#para_solver_residuum#
#para_rho#
#markov_chain_uri#
#markov_series#
#markov_update#
#Lx#
#Ly#
#Lz#
#Lt#
#beta#
#kappa#
#csw#
#today#
#average_plaquette#
#precision#
#crc_check_sum#
#data_lfn#
```

Placeholders are defined in `ildg/ildg_meta.h`. Some placeholders can be determined from the normal input, for others there are special input parameters:

```
ildg_markov_chain_uri            "mc://UNDEFINED"
ildg_data_lfn_path               "lfn://UNDEFINED"
ildg_participant_name            "UNDEFINED"
ildg_participant_institution     "UNDEFINED"
ildg_machine_name                "UNDEFINED"
ildg_machine_institution         "UNDEFINED"
ildg_machine_type                "UNDEFINED"
```

In any case file names for the templates have to be given:

```
ildg_template_ensemble           "ensemble_template.xml"
ildg_template_conf               "config_template.xml"
```

One can also set prefix and extension of the `ildg` files:

```
ildg_filename_prefix              "qcdsf.%R3"
ildg_filename_extension           "lime"
```

This setting would, for example, generate these configuration files:

```
qcdsf.200.xml
qcdsf.200.00010.xml
qcdsf.200.00010.lime
```

3.6.3. Precision

The program can handle `ildg` files in 32- and 64-bit precision. When reading the precision is taken from the `ildg` file. The precision of writing can be set to 32-bit by:

```
ildg_precision          32    SU(3) configurations
ildg_precision_restart  32    restart files (for testing)
```

3.6.4. Example of a complete set of `ildg` settings

```
io_conf_format                "ildg"
ildg_filename_prefix          "qcdsf.%R3"
ildg_filename_extension       "lime"
ildg_precision                 64
ildg_template_ensemble        "../data/qcdsf-ensemble-05.xml"
ildg_template_conf            "../data/qcdsf-configuration-05.xml"
ildg_markov_chain_uri         "mc://ldg/qcdsf/clover_nf2/b5p00kp13000-04x04"
ildg_data_lfn_path            "lfn://ldg/qcdsf/clover_nf2/b5p00kp13000-04x04"
ildg_participant_name         "Hinnerk Stueben"
ildg_participant_institution "Universitaet Hamburg"
ildg_machine_name             "HLRN-III"
ildg_machine_institution      "HLRN"
ildg_machine_type             "Cray XC40"
```

3.7. Output – structure of `res`(ults) file

The output was structured in such a way that it is humanly readable and can be easily processed with *awk*. As a consequence each line begins with a keyword which is followed by data. In addition there are sections. The sections are:

```
>BeginJob
   >BeginHeader
   >EndHeader

   >BeginILDGread
   >EndILDGread

   >BeginForceAcceptance
   >EndForceAcceptance

   >BeginMC
      >BeginILDGwrite
      >EndILDGwrite
      >BeginCooling
      >EndCooling
   >EndMC

   >BeginHMCtest
   >EndHMCtest

   >BeginILDGwrite
   >EndILDGwrite

   >BeginFooter
      >BeginTiming
      >EndTiming
      >Begintiming2
      >Endtiming2
   >EndFooter
>EndJob
```

The Monte-Carlo sections contain tables embedded. A single table can be extracted from the output using *grep* or *awk*, For example:

```
$ grep %mc bqcd.032.res
 T%mc  traj  e  f     PlaqEnergy  exp(-Delta_H) Acc CGcalls CGitTot CGitMax
  %mc     1  1  1   0.4546852837   0.9239828462   0      11     381      37
  %mc     2  1  1   0.4743147510   0.6786574618   1      11     360      34
  %mc     3  1  1   0.4725616177   1.1718570939   1      11     369      34
  %mc     4  1  1   0.4716513122   1.0527896694   1      11     390      36
  %mc     5  1  1   0.4607034796   1.0829331352   1      11     385      35
  %mc     6  1  1   0.4705402342   0.8272890510   1      11     386      36
  %mc     7  1  1   0.4808335455   1.0465054341   1      11     386      36
```

```
%mc      8  1  1   0.4808963557   1.2053582280   1      11      390      36
%mc      9  1  1   0.4808963557   0.7580344698   0      11      417      39
%mc     10  1  1   0.4820214435   0.8795287474   1      11      403      38
```

If one is interested in the values only (without the table header)

```
$ awk '$1 == "%mc"' bqcd.032.res
```

does the job. The general format of a table is:

key trajectory_counter value(s)

Tables introduced at an early stage contain ensemble indices `e` and `f` in addition. This is because a measurement can always belong to two ensembles when working with parallel tempering.

3.7.1. Header section

The header section contains compile (e.g. program version), input (from parameter file) and runtime (e.g. date) parameters. For historic reasons the keywords used for parameters are different from the input file (previous input files only used positional parameters). Another peculiarity are `_1` endings. This ending indicates the ensemble index (what is only necessary for parallel tempering).

3.7.2. ILDG read and write sections

The ILDG sections contain information about the file being read or written, e.g. the filename and the ILDG *logical filename (LFN)*. If a restart file is written in ILDG format the LFN line contains other data, namely the filename, its CRC check-sum, its size in bytes, run-, job- and trajectory counters.

3.7.3. Monte-Carlo sections (`ForceAcceptance`, `MC`, `HMCtest`)

`ForceAcceptance` reports on trajectories that were generated without acceptance test which is sometimes needed at the beginning of a simulation.

`MC` reports on usual Hybrid Monte-Carlo trajectories including optional measurements when a new trajectory is finished.

`HMCtest` reports on a reversibility test. One trajectory is integrated forward and backward. Afterwards energies and fields are compared.

3.7.4. Cooling section

This section contains a cooling history with the measurement of the topological charge.

3.7.5. Footer section

In the footer one finds the end of execution date, the elapsed run time and how many CPUs were used (#CPUs = #cores = MPI processes × OpenMP threads).

3.7.6. Timing sections

Output from profiling. Results are shown in a table that list how often a region was called and how much time was spent in that region:

```
                                        Performance
region      #calls      time        mean       min       max       Total
                           s     Mflop/s   Mflop/s   Mflop/s     Gflop/s
```

For the most interesting regions the number of floating point were counted. In these cases the table lists the average, minimal and maximal performance per CPU (core) as well as the overall performance.

3.7.7. List of embedded tables

key	meaning
`%fa`	forced acceptance logs
`%mc`	main Hybrid Monte-Carlo logs
`%pr`	rectangular plaquettes
`%Hold`	energies at start of trajectory
`%Hnew`	energies at end of trajectory
`%Hdif`	energy differences (ΔH)
`%Favg`	average forces in molecular dynamics integration
`%Fmax`	maximal forces in molecular dynamics integration
`%Frat`	force ratios: maximal forces / minimal forces
`%Frac`	forces in RHMC molecular dynamics integration
`%Qc`	topological charge from cooling (cooling history)
`%pl`	Polyakov loop
`%tr`	fermionic bulk quantities (traces)
`%it`	counters of cg iterations
`%it4`	counters of $\mathrm{cg}_{32\,\mathrm{bit}}$ iterations
`%cgChk`	check of cg solution
`%bicgstabChk`	check of Bicgstab solution
`%bicgstabmixChk`	check of mixed precision Bicgstab solution
`%cg_ritz`	check of Ritz cg solution
`%cg_ritz_dd`	check of Ritz cg solution
`%shift`	check of multi shift solutions
`%egnv`	eigenvalues

3.8. Measurements

Measurements are made at the end of every trajectory.

3.8.1. Topological charge

To measure the topological charge with the cooling method a file must be provided in which the cooling steps are defined. One has to set

`measure_cooling_list` filename

in the input file. The file contains the cooling steps after which the topological charge and the plaquette are measured. For example,

```
1
2
5
10
```

```
20
```

In this case 20 cooling iterations are made and measurements are performed after cooling iteration 1, 2, 5, 10 and 20. The corresponding output looks like this:

```
>BeginCooling
T%Qc  traj  e  f  i_cool        Q_cool     PlaqEnergy
 %Qc     1  1  1       1     -0.137916   0.3552673573
 %Qc     1  1  1       2     -0.347026   0.1878487312
 %Qc     1  1  1       5     -0.650364   0.0555456917
 %Qc     1  1  1      10     -0.533735   0.0181235629
 %Qc     1  1  1      20      0.000154   0.0051610597
>EndCooling
```

3.8.2. Polyakov loop

To measure the Polyakov loop, set

`measure_polyakov_loop` 1

in the input file.

3.8.3. Fermionic bulk quantities

To measure fermionic bulk quantities, set non zero value

`measure_traces` 1

in the input file. The value is number of noises. You can specify the file log of these measurements by `measure_traces_file`. The corresponding outputs are summarized in `%tr`, `%trd` and `%trm`.

Traces up to fourth derivative are labeled as:

$$T_{abcd} = \frac{1}{12N_s^3N_t}\mathrm{Tr}(M_aM_bM_cM_d) \tag{3}$$

where $a, b, c, d = 0, 1, 2, 3$ and

$$\begin{aligned} M_0 &= 1 \\ M_1 &= D^{-1} \\ M_2 &= \left(\frac{\partial D}{\partial \mu}\right) D^{-1} \\ M_3 &= \left(\frac{\partial^2 D}{\partial \mu^2}\right) D^{-1} \end{aligned} \tag{4}$$

By using these labels we write as

$$\left(\frac{\partial}{\partial\mu}\right)^2 \ln\det D = T_{0003} - T_{0022}\,. \tag{5}$$

Traces can be extracted by

```
grep " %tr    "  res-file |awk '{printf("%s %s\n", $4, $5)}' > T0001
grep " %trd   "  res-file |awk '{printf("%s %s\n", $4, $5)}' > T0011
grep " %trd   "  res-file |awk '{printf("%s %s\n", $6, $7)}' > T0111
grep " %trd   "  res-file |awk '{printf("%s %s\n", $8, $9)}' > T1111
grep " %trm01"   res-file |awk '{printf("%s %s\n", $4, $5)}' > T0002
grep " %trm01"   res-file |awk '{printf("%s %s\n", $6, $7)}' > T0003
grep " %trm01"   res-file |awk '{printf("%s %s\n", $8, $9)}' > T0022
grep " %trm02"   res-file |awk '{printf("%s %s\n", $4, $5)}' > T0032
grep " %trm02"   res-file |awk '{printf("%s %s\n", $6, $7)}' > T0033
grep " %trm02"   res-file |awk '{printf("%s %s\n", $8, $9)}' > T0222
grep " %trm03"   res-file |awk '{printf("%s %s\n", $4, $5)}' > T0322
grep " %trm03"   res-file |awk '{printf("%s %s\n", $6, $7)}' > T2222
grep " %trm03"   res-file |awk '{printf("%s %s\n", $8, $9)}' > T0021
grep " %trm04"   res-file |awk '{printf("%s %s\n", $4, $5)}' > T0031
grep " %trm04"   res-file |awk '{printf("%s %s\n", $6, $7)}' > T0211
grep " %trm04"   res-file |awk '{printf("%s %s\n", $8, $9)}' > T0311
grep " %trm05"   res-file |awk '{printf("%s %s\n", $4, $5)}' > T2111
grep " %trm05"   res-file |awk '{printf("%s %s\n", $6, $7)}' > T0221
grep " %trm05"   res-file |awk '{printf("%s %s\n", $8, $9)}' > T0321
grep " %trm06"   res-file |awk '{printf("%s %s\n", $4, $5)}' > T0231
grep " %trm06"   res-file |awk '{printf("%s %s\n", $6, $7)}' > T2211
grep " %trm06"   res-file |awk '{printf("%s %s\n", $8, $9)}' > T2121
grep " %trm07"   res-file |awk '{printf("%s %s\n", $4, $5)}' > T2221
```

3.8.4. Determinants for phase reweighting at non-zero chemical potential

To measure determinants for phase reweighting at non-zero chemical potential at every second trajectory, set

```
measure_chemical   2
```

in the input file. Following 13 lines per one measurement are printed in dethist.mid_*mid*_._*jobcount*_ file.

```
traj    W TrB TrB^2 TrC TrB^3 TrBC TrB^4 TrB^2C TrC^2 log(|A|) arg(A)
traj    W_WNE |W_WNE| arg(W_WNE)
traj  0 V^0
traj  1 V^1
```

```
traj  2 ...
traj  3 ...
traj  4 ...
traj  5 ...
traj  6 ...
traj  7 ...
traj  9 ...
traj 10 ...
```

where `W` is $\det(K)$, `B` is $\dot{K}$, `C` is $\ddot{K}$, `A` is A_0. 2nd line is

$$W_{WNE} = \exp\left[-\sum_{q=1}^{10} (2\cosh(q\mu/T)\mathrm{Re}\hat{V}^{(q)} + i2\sinh(q\mu/T)\mathrm{Im}\hat{V}^{(q)}) \right] \tag{6}$$

and its absolute value and argument. From 3rd line $\hat{V}^{(q)}$ are printed for each trace. Each value is defined in Ref. [40].

3.8.5. f_A and f_P

To measure f_A and f_P for non-perturbative determination of c_{SW} at every second trajectory, set

```
measure_schrpcac   2
```

in the input file. The corresponding output looks like this:

```
traj t+1 fA(t)            fP(t)           fA'(T-t)          fP'(T-t)
   2   2 -0.10939226E+02  0.32266553E+02 -0.56909470E+01  0.17781391E+02
   2   3 -0.34826553E+01  0.87987951E+01 -0.19998378E+01  0.40562673E+01
   2   4 -0.10307930E+01  0.19997526E+01 -0.87475663E+00  0.11811545E+01
   4   2 -0.10427659E+02  0.36927566E+02 -0.57629684E+01  0.18859736E+02
   4   3 -0.49157290E+01  0.13352503E+02 -0.31013816E+01  0.54608054E+01
   4   4 -0.19897425E+01  0.38296835E+01 -0.18940843E+01  0.23258797E+01
```

3.8.6. Wilson flow for QCD field

To measure Wilson flow for QCD field at every second trajectory, set

```
measure_wilson_flow   2
```

in the input file. The corresponding output looks like this (12 digits are printed in actual output):

```
traj, t, Ep(t), Ec(t), Ec(t)t^2, t* dt Ec(t)^2, Qtop, (1-Ep(t))t^2
```

```
0    0.00E+00  0.22E+00  0.22E+01  0.00E+00  0.00E+00 -0.91E-01  0.00E+00
0    0.10E-01  0.25E+00  0.22E+01  0.22E-03  0.22E-03 -0.96E-01  0.74E-04
0    0.20E-01  0.27E+00  0.23E+01  0.93E-03  0.14E-02 -0.10E+00  0.29E-03
0    0.30E-01  0.29E+00  0.23E+01  0.21E-02  0.35E-02 -0.10E+00  0.63E-03
```

where `t` is flow time, `Ep(t)` is plaquette energy, `Ec(t)` is clover energy and `Qtop` is topological charge.

3.8.7. Wilson flow for QED field

To measure Wilson flow for QED field at every second trajectory, set

```
measure_wilson_flow_qed   2
```

in the input file. The corresponding output is same format as Wilson flow for QCD field.

4. Physics

4.1. Gauge actions

The gauge action can be:

- the Wilson action

$$S_G = S_G^{\text{Wilson}} = \beta \sum_{\text{plaquette}} \frac{1}{3} \operatorname{Re} \operatorname{Tr} \left(1 - U_{\text{plaquette}}\right) \tag{7}$$

- an improved gauge action

$$S_G = \frac{6}{g^2} \left[c_0 \sum_{\text{plaquette}} \frac{1}{3} \operatorname{Re} \operatorname{Tr} \left(1 - U_{\text{plaquette}}\right) + c_1 \sum_{\text{rectangle}} \frac{1}{3} \operatorname{Re} \operatorname{Tr} \left(1 - U_{\text{rectangle}}\right) \right], \tag{8}$$

  with $c_0 + 8c_1 = 1$.

Depending on the values of the input parameters `gauge_action` and β = `beta` the program sets g^2, c_0 and c_1 in the following way:

`gauge_action`	g^2	c_0	c_1
`WILSON`	$6/\beta$	1	0
`TREE`	$10/\beta$	1	$-1/20$
`IWASAKI`	$6/\beta$	3.648	-0.331

4.2. Fermionic actions

The fermionic action can be:

- the Wilson action

$$\begin{aligned} S_F^{\text{Wilson}} = \sum_x \Big\{ & \bar{\psi}(x)\psi(x) - \kappa \big[\bar{\psi}(x) U_\mu^\dagger (x - \hat{\mu})(1 + \gamma_\mu)\psi(x - \hat{\mu}) \\ & + \bar{\psi}(x) U_\mu(x)(1 - \gamma_\mu)\psi(x + \hat{\mu}) \big] \Big\} \end{aligned} \tag{9}$$

- the Wilson action plus an explicitly parity-flavour symmetry breaking source term, where τ^3 is the third Pauli matrix

$$S_F = S_F^{\text{Wilson}} + h \sum_x \bar{\psi}(x) i \gamma_5 \tau^3 \psi(x) \tag{10}$$

- the clover $O(a)$ improved Wilson action

$$S_F = S_F^{\text{Wilson}} - \frac{i}{2}\kappa\, c_{\text{SW}} \sum_x \bar{\psi}(x)\sigma_{\mu\nu}F_{\mu\nu}(x)\psi(x) \tag{11}$$

- fat link fermions

$$S_F = \sum_x \Big\{ \bar{\psi}(x)\psi(x) - \kappa\, \bar{\psi}(x)U_\mu^\dagger(x-\hat{\mu})[1+\gamma_\mu]\psi(x-\hat{\mu}) \\ - \kappa\, \bar{\psi}(x)U_\mu(x)[1-\gamma_\mu]\psi(x+\hat{\mu}) + \frac{i}{2}\kappa\, c_{\text{SW}}\, \bar{\psi}(x)\sigma_{\mu\nu}F_{\mu\nu}(x)\psi(x) \Big\}, \tag{12}$$

where the gauge links U_μ are replaced by stout links [28]

$$U_\mu \to \tilde{U}_\mu(x) = e^{iQ_\mu(x)}\, U_\mu(x)\,, \tag{13}$$

with

$$Q_\mu(x) = \frac{\alpha}{2i}\left[V_\mu(x)U_\mu^\dagger(x) - U_\mu(x)V_\mu^\dagger(x) - \frac{1}{3}\text{Tr}\,\left(V_\mu(x)U_\mu^\dagger(x) - U_\mu(x)V_\mu^\dagger(x)\right)\right], \tag{14}$$

where $V_\mu(x)$ is the sum over all staples associated with the link. One can also replace the gauge link by n times smeared links $U_\mu^n(x)$ obtained as

$$U_\mu^n(x) = e^{iQ^{n-1}{}_\mu(x)} U_\mu^{n-1}(x)\,. \tag{15}$$

For SLiNC fermions, $n = 1$ and $\alpha = 0.1$,

```
fermi_action SLRC
csw          2.65    see [10]
n_stout      1
alpha        0.1
```

- the hopping term with chemical potential μ (`chemi`, `chemi_i`)

$$\begin{aligned} &-\kappa \sum_i^3 \left[\bar{\psi}(x)U_i^\dagger(x-\hat{i})(1+\gamma_i)\psi(x-\hat{i}) + \bar{\psi}(x)U_i(x)(1-\gamma_i)\psi(x+\hat{i})\right] \\ &-\kappa \left[\bar{\psi}(x)U_4^\dagger(x-\hat{4})(1+\gamma_4)e^{-\mu}\psi(x-\hat{4}) + \bar{\psi}(x)U_4(x)(1-\gamma_4)e^{\mu}\psi(x+\hat{4})\right] \end{aligned} \tag{16}$$

- the clover $O(a)$ improved Wilson action + CPT breaking term with coefficient λ (`nf2_lambda[1-6]`, `nf1_lambda[1-6]`)

$$S_F = S_F^{\text{Wilson}} - \frac{i}{2}\kappa\, c_{\text{SW}} \sum_x \left[\bar{\psi}(x)\sigma_{\mu\nu}F_{\mu\nu}(x)\psi(x) + \kappa\,\lambda\bar{\psi}(x)H\psi(x)\right], \tag{17}$$

where H is 4×4 matrix specified by `breaking_term` formatted as,

```
------------------------------------------------- breaking_term file
 Re(H11) Im(H11)  Re(H12) Im(H12) Re(H13) Im(H13) Re(H14) Im(H14)
 Re(H21) Im(H21)  Re(H22) Im(H22) Re(H23) Im(H23) Re(H24) Im(H24)
 Re(H31) Im(H31)  Re(H32) Im(H32) Re(H33) Im(H33) Re(H34) Im(H34)
 Re(H41) Im(H41)  Re(H42) Im(H42) Re(H43) Im(H43) Re(H44) Im(H44)
------------------------------------------------- breaking_term file
```

4.3. Schrödinger functional boundary conditions

`boundary_sf` sets Schrödinger functional boundary conditions.

Gauge action is slightly modified with w_0 and w_1,

$$S_G = \frac{6}{g^2}\left[w_0 c_0 \sum_{\text{plaquette}} \frac{1}{3}\,\text{Re}\,\text{Tr}\,(1 - U_{\text{plaquette}}) + w_1 c_1 \sum_{\text{rectangle}} \frac{1}{3}\,\text{Re}\,\text{Tr}\,(1 - U_{\text{rectangle}})\right], \tag{18}$$

$$w_0 = \begin{cases} \frac{1}{2} & \text{all links are on boundary} \\ 1 & \text{otherwise} \end{cases} \tag{19}$$

$$w_1 = \begin{cases} 0 & \text{all links are on boundary} \\ \frac{3}{2} & \text{only two links are on boundary} \\ 1 & \text{otherwise} \end{cases} \tag{20}$$

Spatial links are fixed to

$$\begin{aligned} U(x)|_{x_0=0} &= \exp(aC_k),\ C_k &&= \tfrac{i\pi}{6L_k}\,\text{diag}(-1,0,1)\,, \\ U(x)|_{x_0=T} &= \exp(aC'_k),\ C'_k &&= \tfrac{i\pi}{6L_k}\,\text{diag}(-5,2,3)\,. \end{aligned} \tag{21}$$

All quark fields are zero on boundary.

4.4. QCD+QED

The program can simulate QCD+QED. The following notation is used

$$e^2 = 1/\beta_{\text{QED}}\,, \quad e_q = Q_q e \tag{22}$$

where e is the electron charge, e_q is the electric charge of quark flavour q and

$$Q_u = +2/3\,, \quad Q_d = Q_s = -1/3 \tag{23}$$

for the the up, down and strange quark respectively.

The following actions can be simulated:

$$S = S_G + S_A + \sum_q S_F^q \,. \tag{24}$$

S_G is an SU(3) gauge action, S_A is the non-compact U(1) gauge action

$$S_A = \frac{\beta_{\mathrm{QED}}}{2} \sum_{x,\mu<\nu} \left[A_\mu(x) + A_\nu(x+\hat{\mu}) - A_\mu(x+\hat{\nu}) - A_\nu(x)\right]^2 \,, \tag{25}$$

and the fermion action for flavour q is

$$\begin{aligned} S_F^q \;=\; \sum_x \Bigg\{ & \frac{1}{2} \sum_\mu \Big[\overline{q}(x)(\gamma_\mu - 1) e^{-iQ_q A_\mu(x)} \tilde{U}_\mu(x) q(x+\hat{\mu}) \\ & - \overline{q}(x)(\gamma_\mu + 1) e^{iQ_q A_\mu(x-\hat{\mu})} \tilde{U}_\mu^\dagger(x-\hat{\mu}) q(x-\hat{\mu}) \Big] \\ & + \frac{1}{2\kappa_q} \overline{q}(x) q(x) - \frac{1}{4} c_{\mathrm{SW}} \sum_{\mu,\nu} \overline{q}(x) \sigma_{\mu\nu} F_{\mu\nu}(x) q(x) \Bigg\} \,, \end{aligned} \tag{26}$$

where $\tilde{U}_\mu$ is a singly iterated stout link (see eq. (13)). The clover coefficient c_{SW} was computed non-perturbatively in pure QCD in [38].

The corresponding input keywords are `beta_qed`, `nf2_em_charge[1-6]_x3` and `nf1_em_charge[1-6]_x3`.

4.5. Axion

The program can simulate QCD+Axion. The following actions an be simulated:

$$S = S_G + S_a + \sum_q S_F^q \,. \tag{27}$$

S_a is the scalar action

$$S_a = \kappa_a \sum_x \sum_\mu \Big(\phi_a(x) - \phi_a(x+\mu) \Big) \phi_a(x) \,, \tag{28}$$

where ϕ_a is the axion field. The fermion action for flavour q in the case of Wilson fermions is

$$\begin{aligned} S_F^q = & \sum_x \overline{q}(x) \Big[1 + (\kappa_q \lambda_q + f_{inv} \phi_a) \gamma_5 \Big] q(x) \\ & - \kappa_q \sum_{x,\mu} \Big[\overline{q}(x)(1-\gamma_\mu) U_\mu(x) q(x+a\hat{\mu}) + \overline{q}(x-a\hat{\mu})(1+\gamma_\mu) U_\mu^\dagger(x-a\hat{\mu}) q(x) \Big] \,, \end{aligned} \tag{29}$$

where

$$\begin{aligned} \kappa_q &= \frac{1}{2am_q + 8}\,, \\ \lambda_q &= i2am_q \frac{\theta}{N_f}\,, \\ f_{\text{inv}} &= i2\kappa_q m_q \frac{\sqrt{\kappa_a}}{f_a N_f}\,. \end{aligned} \tag{30}$$

Note: Non-zero values must be specified for κ_a (`kappa_axion`), f_{inv} (`finv_axion`) and γ_5 by `breaking_term`.

4.6. Observables

4.6.1. Gluonic observables

The following gluonic observables can be measured:

- Average plaquette and average rectangular plaquette (both overall, space-like and time-like).
- Topological charge. The topological charge is measured with the field theoretic method after cooling the gauge field configuration.
- Polyakov loop.

4.6.2. Fermionic observables

Some fermionic bulk quantities can be measured (from stochastic estimators):

$$\begin{aligned} \langle\bar{\psi}\psi\rangle &= \frac{1}{12V}\langle \mathrm{Tr}(M^{-1})\rangle \qquad \text{('chiral condensate')} \\ \langle\bar{\psi}\gamma_5\psi\rangle &= \frac{1}{12V}\langle \mathrm{Tr}(\gamma_5 M^{-1})\rangle \\ \langle\Pi^2\rangle &= \frac{1}{12V}\langle \mathrm{Tr}(M^\dagger M)^{-1}\rangle \qquad \text{('pion norm')} \end{aligned}$$

5. Algorithms

5.1. Multi timescale integration

In order to explain multi timescale integration we look at the partition function for N_f=2+1 improved Wilson fermions

$$\begin{aligned} Z &= \int DU D\bar{\psi} D\psi e^{-S}\,, \\ S &= S_g(\beta) + S_l(\kappa_l, c_{\rm SW}) + S_s(\kappa_s, c_{\rm SW})\,, \end{aligned} \tag{31}$$

where S_g is a gluonic action, S_l is an action for the degenerate u- and d-quarks and S_s is an action for the strange quark. After integrating out fermions

$$S = S_g(\beta) - \ln[\det M_l^\dagger M_l][\det M_s^\dagger M_s]^{\frac{1}{2}}\,. \tag{32}$$

We first apply even-odd preconditioning:

$$\det M_l^\dagger M_l \propto \det(1+T_{oo}^l)^2 \det Q_l^\dagger Q_l\,, \quad [\det M_s^\dagger M_s]^{\frac{1}{2}} \propto \det(1+T_{oo}^s)[\det Q_s^\dagger Q_s]^{\frac{1}{2}}\,, \tag{33}$$

where

$$Q = (1+T)_{\rm ee} - M_{\rm eo}(1+T)_{\rm oo}^{-1} M_{\rm oe}\,, \quad T = \frac{\rm i}{2} c_{\rm SW}\,\kappa\,\sigma_{\mu\nu} F_{\mu\nu}\,. \tag{34}$$

We then separate $\det Q_l^\dagger Q_l$ following Hasenbusch [33]

$$\det Q_l^\dagger Q_l = \det W_l^\dagger W_l \det \frac{Q_l^\dagger Q_l}{W_l W_l^\dagger}\,, \qquad W = Q + \rho\,. \tag{35}$$

Finally we modify the standard action to

$$S = S_g + S_{det}^l + S_{det}^s + S_{f1}^l + S_{f2}^l + S_{fr}^s\,, \tag{36}$$

where

$$\begin{aligned} &S_{det}^l = -2\,{\rm Tr}\,\log[1+T_{oo}(\kappa^l)]\,, \quad S_{det}^s = -\,{\rm Tr}\,\log[1+T_{oo}(\kappa^s)]\,, \\ &S_{f1}^l = \phi_1^\dagger [W(\kappa^l)^\dagger W(\kappa^l)]^{-1}\phi_1\,, \quad S_{f2}^l = \phi_2^\dagger W(\kappa^l)[Q(\kappa^l)^\dagger Q(\kappa^l)]^{-1} W(\kappa^l)^\dagger \phi_2\,, \\ &S_{fr}^s = \sum_{i=1}^{n} \phi_{2+i}^\dagger [Q(\kappa^s)^\dagger Q(\kappa^s)]^{-\frac{1}{2n}} \phi_{2+i}\,. \end{aligned} \tag{37}$$

We calculate S_{fr} using the RHMC algorithm [34] with optimised values for n and the number of fractions. We now split each term of the action into one ultraviolet and two infrared parts,

$$S_{\rm UV} = S_g\,, \quad S_{\rm IR-1} = S_{det}^l + S_{det}^s + S_{f1}^l\,, \quad S_{\rm IR-2} = S_{f2}^l + S_{fr}^s\,. \tag{38}$$

In [6] we have introduced two different time scales [35] for the ultraviolet and infrared parts of the action in the leap-frog integrator. Here we shall go a step further and put S_{UV}, $S_{\mathrm{IR}-1}$ and $S_{\mathrm{IR}-2}$ on *three separate* time scales,

$$\begin{aligned} V(\tau) &= \left[V_{\mathrm{IR}-2}\left(\frac{\delta\tau}{2}\right) \quad A^{m_1} \quad V_{\mathrm{IR}-2}\left(\frac{\delta\tau}{2}\right)\right]^{n_\tau}, \\ A &= V_{\mathrm{IR}-1}\left(\frac{\delta\tau}{2m_1}\right) \quad B^{m_2} \quad V_{\mathrm{IR}-1}\left(\frac{\delta\tau}{2m_1}\right), \\ B &= V_{\mathrm{UV}}\left(\frac{\delta\tau}{2m_1m_2}\right) V_Q\left(\frac{\delta\tau}{m_1m_2}\right) V_{\mathrm{UV}}\left(\frac{\delta\tau}{2m_1m_2}\right), \end{aligned} \tag{39}$$

where $\mathrm{n}_\tau = \tau/(\delta\tau)$ and the Vs are evolution operators of the Hamiltonian. The length of the trajectory τ is taken to be equal to one in our simulations.

5.2. Tuning the rational fraction part

BQCD is able to avoid generating coefficients for the rational approximation every time. Specific sets of coefficients are implemented in advance (see code in `fermi/rhmc`). If the approximation range of generated coefficients is wider than the condition number of X, BQCD automatically shifts as follows, when range does not cover actual [min, max] of X

$$\begin{aligned} X^\alpha &= \beta^{-\alpha}(\beta X)^\alpha \\ &\approx \beta^{-\alpha}\left[c_0 + \sum_{i=1} \frac{c_i}{\beta X + d_i}\right] \end{aligned} \tag{40}$$

where β is the inverse of the minimum eigenvalue of X and c_i, d_i is generated by Remez algorithm with range for one to the condition number of X, $[1, C(X)]$.

BQCD also supports to tune the rational approximation by given range and approximation degree in a file

```
tuning_approx_range_list  "rangelist"
```

Example *`rangelist`* file:

```
1 11 15 2
2 10 14 2
```

1st column is ID for rational approximation. It starts from 1 and is consistent with `rid` in `check para` region printed to stderr. 2nd column is degree of approximation used to approximate $1/X^{-n}$ which is used at MD steps. 3rd column is degree of approximation

used to approximate $1/X^{-2n}$ and $1/X^{+2n}$ which are used at action calculation. 4th column is a margin factor of approximation range.

To relax the solver tolerance one can specify

```
tuning_fraction_tolerance  "fractiontolerance"
```

Example *fractiontolerance* file:

```
0.0011
0.55
2.2
0.11
```

In this case, tolerances for 1st, 2nd, 3rd and 4th shift are relaxed by factors of 2000, 4, 1 and 20. This tuning works only if

```
tuning_approx_range  !=0
```

and BQCD is compiled with FMLIB.

5.3. Polynomial filtering

The basic idea of polynomial filtering [15] (PFHMC or PF) is to use a short polynomial of the Dirac matrix M as a UV filter of the fermion action. This has parallels with mass preconditioning, where we use a heavier Dirac matrix as the filter.

5.3.1. Double-flavour case

In the case of a 2-flavour action $S_F = \phi^\dagger (M^\dagger M)^{-1} \phi \equiv \phi^\dagger K^{-1} \phi$, we use a polynomial $P(K)$ of $K = M^\dagger M$ that approximates K^{-1}, and the action becomes

$$S_F = \phi_1^\dagger P(K) \phi_1 + \phi_2^\dagger [P(K)K]^{-1} \phi_2. \tag{41}$$

We can also use multiple such filters. Take two polynomials $P_1(K)$, $P_2(K)$ with orders $p_1 < p_2$ which approximate K^{-1} and factorize into another polynomial $Q(K) = P_2(K)/P_1(K)$ of order q. Then we can use the fermion action

$$S_F = \phi_1^\dagger P_1(K) \phi_1 + \phi_2^\dagger Q(K) \phi_2 + \phi_3^\dagger [P_3(K)K]^{-1} \phi_3. \tag{42}$$

The polynomials in BQCD for the double-flavour case are implemented as Chebyshev polynomials, which take the form

$$P_n(K) = c_n \prod_{i=1}^{n} (K - z_i) \tag{43}$$

with roots

$$z_i = \mu(1 - \cos\theta_k) + i\sqrt{\mu^2 - \nu^2}\sin\theta_k, \quad \theta_k = \frac{2\pi k}{k+1}, \tag{44}$$

and normalization

$$c_n = \left[\mu \prod_{i=1}^{n} (\mu - z_i)\right]^{-1}. \tag{45}$$

Here, μ and ν are two real, positive parameters with $\nu < \mu$.

The roots describe an ellipse in the complex plane, centred at $i\mu$ with semi-major axis μ in the complex direction and semi-minor axis $\sqrt{\mu^2 - \nu^2}$. This ellipse is traversed in an anti-clockwise direction, with θ_k denoting the corresponding angle. μ and ν should be chosen such that the described ellipse sufficiently encapsulates the eigenvalues of the fermion matrix K, as otherwise we reach numerical instabilities.

In the case of multiple filters, we can ensure the factorization of Chebyshev polynomials $P_m(K)$ and $P_n(K)$ $(n > m)$ by fixing (μ, ν) and setting $\mod(n+1, m+1) = 0$.

The polynomials filters can also be read in from file via `nf2_k[1-6]p[1-3]_file`, but the onus is on the user to choose appropriate polynomials. See the corresponding input keyword documentation for information about the required file format. Note that these are read straight into the action term, so (e.g.) `nf2_k1p2_file` is interpreted as $Q(K)$.

5.3.2. Single-flavour case

In the case of a single-flavour action, $S_F = \phi^\dagger R(K)\phi$ where $R(K) \approx K^{-1/2}$ is a rational approximation, we use a polynomial $P(K)$ that approximates $K^{-1/2}$, and the action becomes

$$S_F = \phi_1^\dagger P(K)\phi_1 + \phi_2^\dagger P(K)^{-1} R(K)\phi_2. \tag{46}$$

We can also use multiple such filters: take two polynomials $P_1(K)$, $P_2(K)$ with orders $p_1 < p_2$ where $P_1(K) \approx K^{-1/2}$ and $P_2(K) \approx P_1(K)^{-1}K^{-1/2}$. Then we can use the fermion action

$$S_F = \phi_1^\dagger P_1(K)\phi_1 + \phi_2^\dagger P_2(K)\phi_2 + \phi_3^\dagger [P_1(K)P_2(K)]^{-1} R(K)\phi_3. \tag{47}$$

The single-filter case is implemented in code via a set of fixed Chebyshev approximations to $K^{-1/2}$.

For multiple filters, the polynomials are read from file via `nf1_k[1-6]p[1-3]_file`, which reads polynomials directly into P_1, P_2 as above. See the corresponding input keyword documentation for information about the required file format.

5.4. The generalized multi-scale integration scheme

The generalized integration scheme [16], activated via `hmc_genint`, starts by considering each time-scale having an integrator of their own Hamiltonian $H_i = T + S_i$, where S_i is

the action term on a time-scale. These integrators are then combined with respect to the time update steps T to form an integrator for the full Hamiltonian $H = T + S$. The way this works is to treat the time steps $T(\epsilon)$ as advancing some time parameter from 0 to h over the course of the trajectory, then superimpose the schemes for each H_i onto this axis. A more rigorous explanation of this technique is given in [16], Appendix B.

The advantage of this technique over the usual nested integrator as in section 5.1 is that one has far more freedom in the choice of step-sizes: a step-size only has to neatly factor the trajectory length, rather than each coarser step-size.

When you use generalized integrators in BQCD, the time-scale specifiers `hmc_m_scale` actually refer to the *absolute* number of steps at each scale, rather than (in the nested case) the number of integration steps at this scale per space update step S at the next-coarsest scale. This difference is especially important to note for the improved integrators such as `2MNSTS`, because these have several S updates per integration step. For example, assuming `2MNSTS` integrators, the input keywords

```
hmc_genint 0
hmc_steps  10
hmc_m_scale 1
hmc_m_scale2 2
```

produce the same integrator as

```
hmc_genint 1
hmc_steps  10
hmc_m_scale 20
hmc_m_scale2 80
```

5.5. Truncated RHMC (tRHMC)

Truncated RHMC (tRHMC) is a filtering method for single-flavour pseudo-fermions. We first express the rational approximation of RHMC in the form

$$R(K) = R_{1,n}(K) = c_n \prod_{i=1}^{n} \frac{K + a_i}{K + b_i} \tag{48}$$

with $a_i > a_{i+1}$, $b_i > b_{i+1}$ and $K = M^\dagger M$. If $R_{1,t}(K)$ forms an approximation to target power of K (e.g. $K^{-1/2}$ in the standard case) for all $1 \leq t \leq n$, then we can use this truncated rational approximation as a filter, giving action

$$S_{\text{tRHMC}} = \phi_1^\dagger R_{1,t}(K)\phi_1 + \phi_2^\dagger R_{t+1,n}(K)\phi_2, \tag{49}$$

where we define

$$R_{i,j}(K) = {c_n}^{\delta_{i1}} \prod_{k=i}^{j} \frac{K + a_k}{K + b_k}. \tag{50}$$

The filter term $\phi_1^\dagger R_{1,t}(K)\phi_1$ thus acts as a UV filter here, and can be placed on a finer time-scale.

Currently, tRHMC is only implemented for Zolotarev approximations (`hmc_zolo`). This was chosen because the built-in rational approximation is stored as a sum over poles, which would make implementing the term-splitting as above in tRHMC more difficult. Also, the Zolotarev approximation satisfies the condition that a truncation roughly approximates $1/K$.

Additional filters can be added with increasing truncation order $t_2 > t_1$:

$$S_{\text{tRHMC}} = \phi_1^\dagger R_{1,t_1}(K)\phi_1 + \phi_2^\dagger R_{t_1+1,t_2}(K)\phi_2 + \phi_3^\dagger R_{t_2+1,n}(K)\phi_3. \tag{51}$$

5.6. The Zolotarev optimal rational approximation

The Zolotarev optimal rational approximation approximates the inverse square-root $R(K) \approx K^{-1/2}$, and is constructed from elliptic functions. It can be shown that this approximation minimizes the error δ, defined as the maximum difference between $R(K)$ and $K^{-1/2}$ on the given range, for the set of all rational approximations with the same order and range, and is thus 'optimal'. See e.g. [39] for more information.

Refer to the keyword documentation in section 3.3.8 for information on how to use this approximation in BQCD.

6. Implementation issues

In this section we explain why things in BQCD are the way they are.

6.1. Programming language

BQCD is mainly written in Fortran. The reasons for this decision were the following. First of all, the programmer was a Fortran programmer. C was not chosen because it was not plausible to write a program that used complex arithmetic almost throughout in a language that had no support for that (or better to say only had added complex arithmetic in C99). C++ was considered but the feeling was that at least a first implementation would have been completed before it would be understood how to use C++ effectively.

The main disadvantages of this decision became visible when the program became more and more complex. Fortran90 eventually supported dynamical memory management but there was no support for dynamical algorithms, i.e., there were no function pointers (procedure pointers available in Fortran2003). The second disadvantage is that working with classes instead of arrays would offer new possibilities like introducing different orderings of the lattice sites which is an interesting optimisation option (of course this can be done in the current approach but the program will become less readable).

One design goal was to write readable code. We have to leave to the reader to check how well this was achieved.

6.2. Preprocessing

6.2.1. C preprocessor

The C preprocessor was employed from the beginning. It is used for conditional compilation and for macro processing. By convention all BQCD source files have the suffix `.F90`. The suffix of the preprocessed file is `.f90`. In some cases several `.f90` files are being generated from a `.F90` file. The `.f90` file are being compiled. They are always kept such that one can always check the result of preprocessing.

Macro names are all uppercase (sometimes mixed case). Fortran code is always lowercase.

There are macros with and without arguments. Macros without arguments are used for defining constants and datatypes. The motivation for this was mainly readability (and aesthetics we have to admit), compare 'BQCD style'

```
# include "defs.h"
```

```
GAUGE_FIELD :: u
COMPLEX :: x, y

...
x = TWO * y + u(...)
```

with a pure Fortran style:

```
use defs

type(gauge_field) :: u
complex(rkind) :: x, y

...
x = const%two * y + u%u(...)
```

Macros with arguments are used as tools, e.g. the `ASSERT` and `ALLOCATE` macros and for simplifying programming. When marcos are used for the latter purpose it is a good idea to look at the `.F90` and the generated `.f90` files when studying the source code.

Over the years C preprocessors became more picky. For example, the GNU preprocessor now refuses to process general Fortran90 code. In some situations C preprocessors complain about the Fortran `%` character (which is a separator in Fortran but an operator in C) and the dots in Fortran operators like `.or.` (because dots are separators in C). In some situations it became necessary to introduce a second preprocessing step, see `tool/fpp_step2`.

The latest change was the need to add `--sysroot=.` to `cpp` calls (otherwise C style comments from some standard C include files would appear in the Fortran code).

6.2.2. m4 macro preprocessor

The *m4* macro processor is also used. In this case there are two preprocessing steps: first a `.F90` file is generated and then a `.f90` file.

6.2.3. `loopp` loop preprocessor

In order to facilitate generic programming with SIMD intrinsics a loop preprocessor, called `loopp`, was written. For example, it would expand

```
DO K = 1, 2
DO J = 1, 2
DO I = 1, 2
```

```
      c%%I%%J = a(I, K) * b(K, J) + c%%I%%J
    ENDDO
    ENDDO

    ENDDO
```

to:

```
    c11 = a(1, 1) * b(1, 1) + c11
    c21 = a(2, 1) * b(1, 1) + c21
    c12 = a(1, 1) * b(1, 2) + c12
    c22 = a(2, 1) * b(1, 2) + c22

    c11 = a(1, 2) * b(2, 1) + c11
    c21 = a(2, 2) * b(2, 1) + c21
    c12 = a(1, 2) * b(2, 2) + c12
    c22 = a(2, 2) * b(2, 2) + c22
```

6.3. Fortran modules

Modules are used for storing global data, type definitions and a few interface definitions. Global data is always readonly except for its initialisation (there are a few exceptions to this rule). In general, modules do not contain functions or subroutines. The idea behind this is that it should always be possible to call functions or subroutines from C/C++ if it should become necessary to do so. Modules that are only used within the same file are put into that file. Modules that are used by more than one file are put into the `modules` subdirectory.

6.4. Precision

Also from the beginning it was foreseen that one might be interested in multi-precision code. In principle one can compile any version of BQCD using single precision arithmetic if one defines:

```
#define RKIND 4
#define BQCD_REAL mpi_real4
```

This feature was used much later to generate multi-precision code. The recipe is the following.

- The original source file, `foo.F90` say, is compiled as usual with double precision arithmetic.

- A single precision version `foo_r4.F90` is generated. It contains only four lines:

```
#define PRECISION_R4
#include "defs.h"
#include "defs_r4.h"
#include "foo.F90"
```

- `defs_r4.h` contains macros for renaming all subroutines and functions, e.g.:

```
#define fun1 fun1_r4
#define fun2 fun2_r4
```

Again there is also a Fortran way of handling the multi-precision problem. One can use interfaces and overloading (see `su3sc/module_sc.F90`).

6.5. Parallelisation

The early versions of BQCD were parallelised by using the *shmem* library from Cray. Later an MPI version was added and also a single processor version that can be compiled without any message passing library. Currently only the MPI and single processor version work (the routines using *shmem* are still contained in the distribution). All files that use calls to message passing routines are located in directory `comm`. Which message passing library to use can be selected in `Makefile.var`.

BQCD is parallelised with OpenMP in addition. On the Hitachi SR8000 this lead to great performance by overlapping communication and computation. In order to facilitate OpenMP programming, BQCD routines contain typically only one loop. It is then straightforward to add OpenMP private and reduction declarations: the candidates can be found in the type declarations of the routine (`implicit none` is used throughout).

The parallel design is such that results are independent of the numbers of processes used. This is true up to rounding errors introduced by global summations. In a job chain one can change the number of processes in every job. This feature was implemented, of course, in order to be able to adapt to changing job mixes at computer centres.

6.6. Random numbers

The first random number generator used was *ranf* by Cray because it has the ability to jump to an arbitrary position in the sequence of random numbers and this operation is not much more expensive than generating the next random number. This skipping of random numbers was reversely engineered such that it was also available on other computers. Skipping is used to generate distributed random numbers in such a way that results become independent of the number of processes.

For running production *ranlux* [36, 37] is the recommended random number generator. In the beginning, the same skipping mechanism as for *ranf* was used. As a consequence

generation of random numbers was not parallelised (random numbers did not have to be communicated, but rather every process generated all random numbers, and only picked the ones belonging to its local lattice). Up a moderate number of parallel processes the overall performance loss is acceptable.

For running on very many processes, parallel generation of *ranlux* numbers is available. There is one random number generator per (x, y)-plane. On each plane the skipping mechanism described above is used. In total there are $L_z \times L_t$ instances of *ranlux*. The size of the state of all random number generators is only $L_z \times L_t \times 105$ integers which can easily be stored with every configuration (when using one generator per lattice site, storage requirements would roughly double). In order to use parallel random number generation `tuning_ran_field_parallel` has to be set to `1`.

6.7. Saving and reading configurations

Internally BQCD differentiates between restart files and files that are supposed to be saved. However, the file structures are the same. Every set of files can be used for restarting a job chain. They contain all internal states (of counters and random number generators).

6.7.1. I/O format `bqcd`

BQCD's file format for configurations was designed to enable simple parallel I/O. Metadata is kept separate (`.info` files) from binary data (`.u` files). To enable parallel I/O one binary file is written for each timeslice. Again, the design is such that everything works on any number of processes. All binary data is written to disk in big endian format. BQCD automatically converts to little endian if necessary. Only two columns of SU(3) matrices are stored. Checksums are calculated on the fly and added to the metadata. Checksums can be verified with standard *cksum* command. This file format is called `bqcd`.

6.7.2. I/O format `bqcd2`

The `bqcd2` format differs from the `bqcd` format in that it writes binary data to a single file using MPI-I/O. Again, metadata is stored in `.info` files. Binary data is stored in `.su3` files.

6.7.3. I/O format `ildg`

Alternatively *lime* files can be written conforming to the ILDG standard [26]. Restart files are always written in 64 bit precision. Configurations can be saved in 32 or 64 bit precision. This kind of I/O is not parallelised.

6.8. Performance measurements and profiling

A simple profiling mechanism was built in. It can be switched on by defining the `TIMING` macro. If it is switched off there is no overhead. In the most important routines operations were counted manually in order to get performance figures.

6.9. Fermionic boundary conditions

BQCD started with having only one copy of the gauge field. Fermionic boundary conditions were imposed by multiplying SU(3) links with -1 accordingly. Flipping boundary conditions between gluonic and fermionic is handled by subroutine `flip_bc()`. It has to be called before and after fermionic operations.

One can optimise the hopping matrix multiplication by introducing a copy of the gauge field that has an optimised storage ordering. This copy then has fermionic boundary conditions (and might also include factors 2 from the $(1 \pm \gamma_4)$ projection).

6.10. C interface

C routines have to be called here and there. Examples are checksum calculations, *ranlux* random numbers and I/O of *lime* files. The Fortran name mangling scheme is selected by defining `NamesToLower`, `NamesToLOwer_` or `NamesToLower__`, respectively.

6.11. Input parsing

The input parser checks whether keywords are known but does not check the rest of the line! It is easy to add a new keyword. New keywords can be introduced by adding them to `modules/module_input.h`. A similar mechanism was used to introduce placeholders for ILDG metadata files (see `ildg/ildg_meta.F90`).

7. Compute performance tuning

Several parameters that affect compute performance can be set at compile- or run-time.

Up to version 4 of BQCD performance tuning concentrated on the hopping matrix multiplication (see section 7.2). Highest optimizations were achieved by Thomas Streuer who added assembler routines for this operation for IBM BlueGene/L and -/P as well as for SGI Altix 4700. The version of the hopping matrix multiplication to be used was specified at compile time.

The current version includes optimized routines that are programmed with SIMD intrinsics (cf. appendix D). The extent of optimization with SIMD is wider: now whole solvers are being worked on (see the following section).

7.1. Conjugate gradient solvers and SIMD vectorization

SIMD vectorization requires to rethink and recode large parts of the basic implementation. For a portable SIMD implementation the array layout has to be adapted (see appendix D and section 7.3.3). This implies that also neighbour lists and communication routines have to be changed accordingly.

Changing the array layout in the whole program would be too demanding. On the other hand, restricting the new layout to the hopping matrix multiplication would generate too much overhead, because input and output arrays would have to be reordered between the old and new layout in every call. However, reordering arrays at the beginning and end of a whole solver is acceptable.

In the new code many variants and implementations can be chosen an run-time. The main choices can be made by setting the following input parameters (see also section 3.3.21):

`tuning_cg_version`	use old or new/SIMD version (`1` or `2`)
`tuning_cg_d`	implementation of the hopping matrix multiplication, see section 7.2
`tuning_cg_simd`	use SIMD or not (`1` or `0`)
`tuning_cg_spincol`	layout of spin-colour arrays, see section 7.3.1
`tuning_cg_clover`	layout of clover arrays, see section 7.3.2

7.2. Hopping matrix multiplication

The most time consuming part of a QCD program is the *hopping matrix* multiplication (a.k.a. *d-slash* operation). Over time several optimizations of this operation were implemented. There are high and low level optimizations. High level are optimizations that can be implemented in the usual programming language while low level are methods like programming in assembler or with SIMD intrinsics.

High level code optimizations in BQCD include:

Loop over directions μ. This is always the outer loop. The inner loop runs over lattice sites. In the very first implementation the forward and backward directions $\pm\mu$ were separate loops. Now the is one inner loop per μ.

Projections $(1 \pm \gamma_\mu)\psi(x \mp \hat{\mu})$. Projections reduce 4-component spinors to 2-component spinors. Projections can be made before or after the exchange of halo regions. One can project at all sites or only at process-boundaries. With our choice of γ-matrices projection in t-direction involves no computation (it is just a selection of the upper or lower two spinor components).

Overlapping communication and computation. See section 7.4.2

Version	Description	Availability with *cg* version (see section 7.1)	
		1	**2**
1	First version (on Cray T3E). There are seven loops: x-forward, x-backward, y-forward, y-backward, z-forward, z-backward, t forward and backward	×	
2	Four loops (loop fusion of the x, y, z forward/backward loops).	×	×
21	Version 2 plus reduction of MPI traffic by projecting at *all* sites before the halo exchange.	×	×
25	Version 2 plus reduction of MPI traffic by projecting only at *boundary sites* sites before the halo exchange (minimal number of projections)		×
3	Version 2 plus overlapping communication and computation.	×	
35	Version 25 plus overlapping communication and computation.		×
100	Version 21 with a customized copy of the gauge field.	×	

Table 1: Implementations of the hopping matrix multiplication. The version is specified at compile time for *cg* version 1. For *cg* version 2 it can by chosen at run-time by setting input parameter `tuning_cg_d`.

Descriptions of the implementations of the hopping matrix multiplication are listed in table 1. Version 2 is the fastest implementation in pure OpenMP mode (single process). Version 3 was the fastest implementation for the Hitachi SR8000, with version 21 TFlop/s were sustained on the IBM BlueGene/L and the SGI Altix 4700, and version 100 was even a bit faster. Version 25 (single threaded) is the fastest on Cray XC30/40 and version 35 on IBM BlueGene/Q (3 or 4 symmetric threads per MPI process).

7.3. Array layout

7.3.1. Spin-colour arrays

There are two layouts for spin-colour arrays that can be used if `tuning_cg_version` is 2:

- `tuning_cg_spincol = 1`

  ```
  complex(8), dimension (4, 3, volume/2 + boundary)
  ```

  This layout that is also used in the rest of the program.

- `tuning_cg_spincol = 22`

  ```
  complex(8), dimension (2, 3, volume/2 + boundary, 2)
  ```

  For the set of γ-matrices that is used in BQCD (see appendix A) this layout leads to minimal overhead in the exchange of boundary values in t-direction: input and output buffers are both consecutive blocks in memory.

7.3.2. Clover arrays

There are two layouts for clover arrays that can be used if `tuning_cg_version` is 2:

- `tuning_cg_clover = 1`

  A packed format (see the source code, file `modules/typedef_clover.F90`).

  This layout that is also used in the rest of the program.

- `tuning_cg_clover = 2`

  A packed format that is supposed to be better suited for SIMD (see the source code, file `cg/typedef_cg_clover.F90`).

Both formats can be used with SIMD and without.

7.3.3. Array layout for SIMD vectorization

If `tuning_cg_simd` is `1` the original array layout

```
complex(8) :: x(dim1, dim2, (volume/2 + boundary), ...)
```

is changed to

```
real(8) :: x(B, re:im, dim1, dim2, (volume/2 + boundary) / B, ...)
```

where `B` is the SIMD register width (see appendix D).

7.4. MPI communication

On the first machine that BQCD was running on in production (Cray T3E) it was advantageous to exchange boundary values direction by direction. For that machine the boundary exchange was implemented with `MPI_Sendrecv`.

7.4.1. Overlapping communications

On later and today machines it is advantageous to exchange all boundaries in an overlapping way. This is implemented with `MPI_Irecv`, `MPI_Isend` and `MPI_Waitall`

```
do n = 1, n_directions
   call mpi_irecv(n)
   call mpi_isend(n)
enddo
call mpi_waitall(n_directions)
```

where the full parameter lists of MPI calls are omitted.

This communication pattern is used by default. It can be switched back to `MPI_Sendrecv` for the spin-colour boundary exchange if `tuning_xbound_sc_i` ist set to `0` and for the gauge field if `tuning_xbound_g_i` ist set to `0`.

7.4.2. Overlapping communication and computation

Overlapping communication and computation is implemented for the boundary exchange of the spin-colour arrays in the hopping matrix multiplication. The algorithm employed is a two-stage *pipeline*:

	communicate direction	compute direction
step 1	y	x
step 2	z	y
step 3	t	z
step 4		t

Technically this is implemented by a *hybrid* parallelization using MPI plus OpenMP. The *main thread* calls MPI while the other threads compute. At least 2 OpenMP-Threads per MPI process are required.

For *cg* version 1 overlapping communication and computation is implemented in version 3 of the hopping matrix multiplication, for *cg* version 2 in version 35 of the hopping matrix multiplication (see table 1).

7.5. Parallel random numbers

Parallel random number generation is described in section 6.6 (`tuning_ran_field_parallel` has to be set to `1`).

7.6. I/O

I/O formats are described in section 6.7. Considering performance `bqcd` is the fastest. `bqcd2` is roughly 30 % slower. Both formats use parallel I/O. `ildg` is not parallel and considerably slower. In practice on can use one of the `bqcd` formats and convert saved configurations to the `ildg` format in a second step by using the `--convert-to-ildg` command line option (see section 3.2).

7.7. Miscellaneous

There are several experimental input parameters for further performance tuning, which are listed here for completeness:

`tuning_cg_precision_r4`
`tuning_cg_precision`
`tuning_cg_loop_blocking`
`tuning_cg_block_length`
`tuning_cg_mcg_block_length`
`tuning_io_ildg`

A. γ-matrix definitions

BQCD uses the following set of gamma matrices:

$$\gamma_1 = \begin{pmatrix} 0 & 0 & 0 & +i \\ 0 & 0 & +i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}$$

$$\gamma_2 = \begin{pmatrix} 0 & 0 & 0 & +1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ +1 & 0 & 0 & 0 \end{pmatrix}$$

$$\gamma_3 = \begin{pmatrix} 0 & 0 & +i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & +i & 0 & 0 \end{pmatrix}$$

$$\gamma_4 = \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

$$\gamma_5 = \begin{pmatrix} 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & +1 \\ +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \end{pmatrix}$$

B. Preprocessor flags – MYFLAGS

This appendix we list important preprocessor flags that are set via MYFLAGS in Makefile.var and Makefile.in. Makefile.var has to be adjusted to new platforms while usually Makefile.in needs no modification.

B.1. Flags set in Makefile.var

ALIGNMENT_VALUE	Value in bytes for aligned memory allocation (must fit to USE_SIMD_* chosen).
CRAY	Settings for the Cray compiler.
IBM	Settings for the IBM compiler.
GNU	Settings for the GNU compiler.
INTEL	Settings for the Intel compiler
I_TIMES_MACRO	At many places a statement function i_times(z) is used. It returns z multiplied by the imaginary unit. (No floating point operations are necessary here.) Because statement functions are outmoded one can achieve the same by using a preprocessor macro.
LongLong	For C source files: 8 byte integers are long long.
NamesToLower NamesToLower_ NamesToLower__	For C source files: set Fortran name mangling scheme, lower case with no, one or two underscores appended.
_OPENMP	Is set automatically when compiling C with OpenMP. Needs to be added for preprocessing Fortran (is now added to FPP rather than MYFLAGS).
TIMING	Switch profiling on (time and performance measurement).
TRACE	Switch on tracing (applies to code in the cg directory).
USE_MPI_WTIME	Use MPI_Wtime for time measurements.
USE_SIMD_GENERIC USE_SIMD_SSE USE_SIMD_AVX USE_SIMD_AVX2 USE_SIMD_QPX	Chose SIMD implementation. GENERIC means plain Fortran instead of SIMD intrinsics. ALIGNMENT_VALUE has to be set accordingly.

B.2. Flags set in Makefile.in

DOEDEO	If not defined (default) the even/odd ordered hopping matrices appear in the preconditioned matrix as $D_{\mathrm{eo}}D_{\mathrm{oe}}$ and as $D_{\mathrm{oe}}D_{\mathrm{eo}}$.
GAMMA_NOTATION_BQCD	γ-matrices are defined according to appendix A.
GAMMA_NOTATION_CHROMA GAMMA_NOTATION_CHIRAL GAMMA_NOTATION_DDHMC GAMMAC	Alternative conventions for γ-matrices.
OMTDTD	If not defined the preconditioned clover improved fermion matrix reads $M = T_{\mathrm{ee}} - D_{\mathrm{eo}}T_{\mathrm{oo}}^{-1}D_{\mathrm{oe}}$ and $M = 1 - T_{\mathrm{ee}}^{-1}D_{\mathrm{eo}}T_{\mathrm{oo}}^{-1}D_{\mathrm{oe}}$ otherwise.

C. Process mapping

Process mapping is the mapping of MPI *ranks* to MPI process coordinates. The functionality is similar to *Cartesian topologies* in MPI.

The following notation is used:

`rank`	MPI rank
`coord()`	process coordinates
`processes()`	values of `processes`, see section 3.3.2
`process_mapping()`	values of `process_mapping`, see section 3.3.2

The standard process mapping corresponds to the mapping of the indices of multi-dimensional array to a single index (expressed in Fortran):

```
rank = 0
rank = rank * processes(4) + coord(4)
rank = rank * processes(3) + coord(3)
rank = rank * processes(2) + coord(2)
rank = rank * processes(1) + coord(1)
```

Through `process_mapping` the mapping corresponds to a mapping of permutated indices:

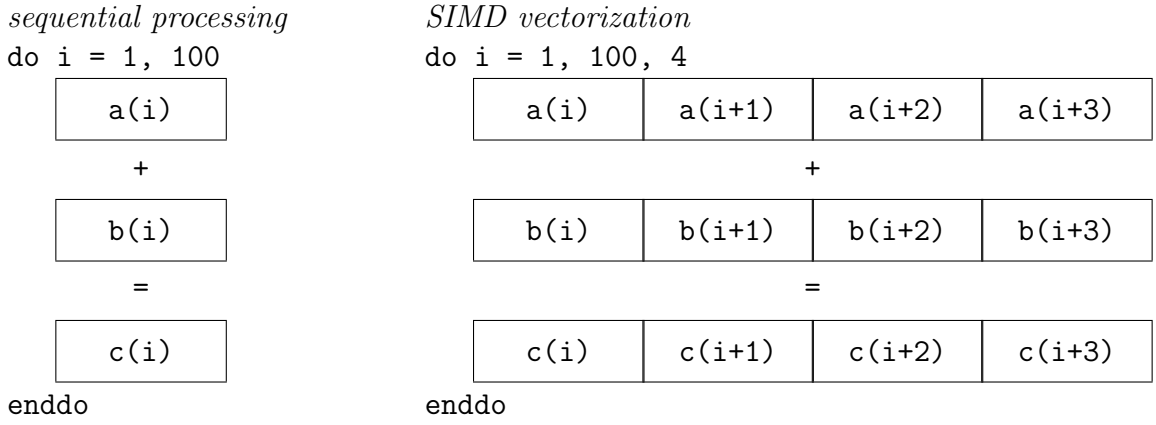
```
rank = 0
rank = rank * processes(process_mapping(4)) + coord(process_mapping(4))
rank = rank * processes(process_mapping(3)) + coord(process_mapping(3))
rank = rank * processes(process_mapping(2)) + coord(process_mapping(2))
rank = rank * processes(process_mapping(1)) + coord(process_mapping(1))
```

D. SIMD vectorization

SIMD stands for *single instruction, multiple data*. SIMD vectorization is an important optimization on modern processors. In SIMD processing mode data is being processed in small groups called SIMD vectors. The vector length is hardware-dependent. In the following illustration we assume that the SIMD vector length, or the width of a vector register, is 4. The loop

```
do i = 1, 100
   c(i) = a(i) + b(i)
enddo
```

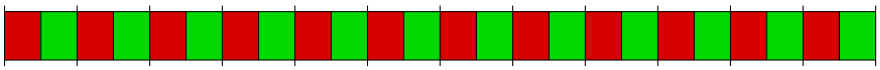
can be processed sequentially or SIMD vectorized:



It is important to note that the multiplication of two Fortran complex numbers is not SIMD, i.e. it cannot be expressed in a way that corresponds to the illustration given above. In order to implement SIMD vectorization the structure of complex arrays has to be changed from *standard* to, for example, one of the other layouts shown below. In BQCD the *SIMD vectors* layout is used (see also section 7.3.3).

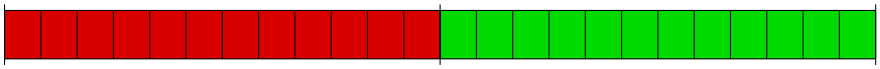
- standard – "array of structs"

  `z(re:im, n)`



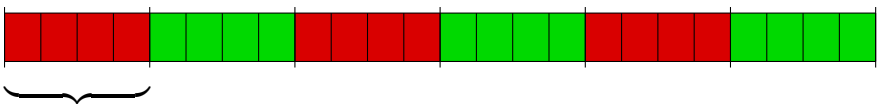
- vector computers – "struct of arrays"

  `z(n, re:im)`



- SIMD vectors

  `z(B, re:im, n/B)`



B: width of a SIMD register

SIMD vectorization can be achieved automatically by compilers or can manually be implemented by employing *intrinsic SIMD functions*. A generic implementation must be independent of the SIMD register width and use only common intrinsics. Common intrinsics available with Intel and IBM BlueGene compilers are:

intrinsic (generic name)	explanation
load	load vector data into a SIMD register
load_scalar	fill SIMD register with a scalar
store	store vector data from a SIMD register
add	`a(:) + b(:)`
sub	`a(:) - b(:)`
mul	`a(:) * b(:)`
div	`a(:) / b(:)`
muladd	`a(:) * b(:) + c(:)`
mulsub	`a(:) * b(:) - c(:)`

E. Loop blocking

Loop blocking is an optimization technique for loops. For example,

```
do i = 1, n                          do i0 = 1, n, 256
   a(i) = b(i) + c(i)                   i1 = min(i0 + (256-1), n)
enddo                                
                                        do i = i0, i1
                                           a(i) = b(i) + c(i)
                                        enddo
                        ⟶
do i = 1, n                             do i = i0, i1
   x(i) = a(i) * y(i)                      x(i) = a(i) * y(i)
enddo                                   enddo

                                     enddo
```

where the *block length* is `256` and it is assumed that `mod(n, 256) == 0`. The idea of this optimization is to improve data cache locality (i.e. to have `a(i)` for the second loop in the data cache after blocking).

References

[1] Y. Nakamura and H. Stüben, PoS(Lattice 2010)040.

[2] T.R. Haar, Y. Nakamura and H. Stüben, Poster presentation at Lattice 2017.

[3] E. M. Ilgenfritz, W. Kerler and H. Stüben, Nucl. Phys. Proc. Suppl. **83** (2000) 831, [arXiv:hep-lat/9908022].

[4] E. M. Ilgenfritz, W. Kerler, M. Müller-Preussker and H. Stüben, Phys. Rev. D **65** (2002) 094506 [arXiv:hep-lat/0111038].

[5] H. Stüben [QCDSF-UKQCD Collaboration], Nucl. Phys. Proc. Suppl. **94** (2001) 273, [arXiv:hep-lat/0011045].

[6] A. Ali Khan, T. Bakeyev, M. Göckeler, R. Horsley, D. Pleiter, P. Rakow, A. Schäfer, G. Schierholz, H. Stüben [QCDSF Collaboration], Phys. Lett. B **564** (2003) 235 [arXiv:hep-lat/0303026].

[7] A. Ali Khan, T. Bakeyev, M. Göckeler, R. Horsley, D. Pleiter, P.E.L. Rakow, A. Schäfer, G. Schierholz, H. Stüben [QCDSF Collaboration], Nucl. Phys. Proc. Suppl. **129** (2004) 853 [arXiv:hep-lat/0309078].

[8] M. Göckeler *et al.* [QCDSF Collaboration], PoS **LAT2007** (2007) 041 [arXiv:0712.3525 [hep-lat]].

[9] N. Cundy *et al.* [QCDSF-UKQCD Collaborations], PoS **LATTICE2008** (2008) 132 [arXiv:0811.2355 [hep-lat]].

[10] N. Cundy *et al.*, Phys. Rev. D **79** (2009) 094507 [arXiv:0901.3302 [hep-lat]].

[11] W. Bietenholz *et al.* [QCDSF-UKQCD Collaborations], PoS **LAT2009** (2009) 102 [arXiv:0910.2963 [hep-lat]].

[12] X.-Y. Jin, Y. Kuramashi, Y. Nakamura, S. Takeda, A. Ukawa, Phys. Rev. D88 (2013) 094508, [arXiv:1307.7205 [hep-lat]] and Phys. Rev. D92 (2015) 114511, [arXiv:1504.00113 [hep-lat]].

[13] A. J. Chambers et al, Phys. Rev. D90 (2014) 014510, [arXiv:1405.3019 [hep-lat]] and Phys. Rev. D92 (2015) 114517, [arXiv:1508.06856 [hep-lat]]

[14] R. Horsley et al., JHEP 1604 (2016) 093, [arXiv:1509.00799 [hep-lat]].

[15] W. Kamleh and M. Peardon Comput. Phys. Comm. **183** (2012) 1993 [arXiv:1106.5625 [hep-lat]]

[16] T. Haar, W. Kamleh, J. Zanotti, Y. Nakamura, Comput. Phys. Commun. 215 (2017) 113, [arXiv:1609.02652 [hep-lat]].

[17] G. Schierholz and Y. Nakamura, Talk by G.S. at Lattice 2017.

[18] E. M. Ilgenfritz, W. Kerler, M. Müller-Preussker, A. Sternbeck and H. Stüben, Phys. Rev. D **69** (2004) 074511, [arXiv:hep-lat/0309057].

[19] A. Sternbeck, E. M. Ilgenfritz, W. Kerler, M. Müller-Preussker and H. Stüben, Nucl. Phys. Proc. Suppl. **129** (2004) 898 [arXiv:hep-lat/0309059].

[20] Y. Nakamura *et al.*, AIP Conf. Proc. **756** (2005) 242 [Nucl. Phys. Proc. Suppl. **140** (2005) 535] [arXiv:hep-lat/0409153].

[21] V. G. Bornyakov *et al.*, arXiv:0910.2392 [hep-lat].

[22] H. Baier *et al.*, arXiv:0911.2174 [hep-lat].

[23] G.S. Bali, S. Collins, A. Cox, A. Schäfer, [arXiv:1706.01247v1 [hep-lat]].

[24] X.-Y. Jin, Y. Kuramashi, Y. Nakamura, S. Takeda, A. Ukawa, [arXiv:1706.01178 [hep-lat]].

[25] S. Hollitt, P. Jackson, R. Young, J. Zanotti, PoS(INPC2016) 272.

[26] `http://ildg.sasr.edu.au/Plone`

[27] K. Symanzik, Nucl. Phys. **B226** (1983) 187.

[28] C. Morningstar and M. J. Peardon, Phys. Rev. **D69** (2004) 054501 [hep-lat/0311018].

[29] S. Capitani, S. Dürr and C. Hoelbling, JHEP 0611 (2006) 028 [hep-lat/0607006].

[30] H. Perlt *et al.* [QCDSF Collaboration], PoS(LATTICE 2007)250 [arXiv:0710.0990].

[31] S. Boinepalli *et al.*, Phys. Lett. **B616** (2005) 196 [hep-lat/0405026].

[32] J.M.Zanotti *et al.*, Phys. Rev. **D71** (2005) 034510 [hep-lat/0405015].

[33] M. Hasenbusch, Phys. Lett. **B519** (2001) 177 [hep-lat/0107019].

[34] M. A. Clark and A. D. Kennedy, Nucl. Phys. Proc. Suppl., **129** (2004) 850 [hep-lat/0309084].

[35] J. C. Sexton and D. H. Weingarten, Nucl. Phys. **B380** (1992) 665.

[36] M. Lüscher, Comput. Phys. Commun. **79** (1994) 100 [arXiv:hep-lat/9309020].

[37] `http://luscher.web.cern.ch/luscher/ranlux/index.html`

[38] N. Cundy, M. Göckeler, R. Horsley, T. Kaltenbrunner, A. D. Kennedy, Y. Nakamura, H. Perlt, D. Pleiter, P. E. L. Rakow, A. Schäfer, G. Schierholz, A. Schiller, H. Stüben and J. M. Zanotti [QCDSF-UKQCD Collaboration] Phys. Rev. D **79** (2009) 094507 [arXiv:0901.3302 [hep-lat]].

[39] T. Chiu, T. Hsieh, C. Huang, and T. Huang, Phys. Rev. D **66** (2002) 114502 [arXiv:hep-lat/0206007].

[40] S. Takeda, Y. Kuramashi, A. Ukawa, Phys. Rev. D **85** (2012) 096008 [arXiv:1111.6363 [hep-lat]].